Departamento de
Universidade de Aveiro Electrónica, Telecomunicações e Informática,
2008

Frederico Valente

Análise estática em sistemas heterogéneos multiprocessador embutidos

Static analysis on embedded heterogeneous multiprocessor systems

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2008

Frederico Valente

Análise estática em sistemas heterogéneos multiprocessador embutidos

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestrado Integrado em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Luís Filipe de Seabra Lopes, Professor Auxiliar do Departamento de Electronica e Telecomunicações da Universidade de Aveiro e Eng. Orlando Moreira da NXP Semiconductors.

o júri / the jury

presidente / president

Doutor António Ferreira Pereira de Melo
Professor Catedrático da Universidade de Aveiro

vogais / examiners committee

Doutor Luís Filipe de Seabra Lopes
Professor Auxiliar da Universidade de Aveiro (orientador)

Doutor Joaquim José Castro Ferreira
Professor Adjunto do Departamento de Engenharia Informática da Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco

agradecimentos / acknowledgements

Aproveito esta oportunidade para agradecer a todos aqueles que me apoiaram durante o meu percurso universitário. Em particular, esta tese não teria lugar não fosse o apoio e ajuda dos meus orientadores Orlando Moreira e Luis Seabra Lopes. E como apenas trabalho não chega para fazer uma tese, ficam também os meus agradecimentos aos meus colegas Bruno Lopes, Miguel Malheiro, Pue, Helder e Claudio, e a todos aqueles amigos que dela me subtrairam quando mais falta fazia (e menos jeito dava). Como não poderia deixar de ser, aos meus pais e à minha irmã, por tudo.

palavras-chave Escalonamento, sistema multi-processador, sistema embutido, mapeamento, tempo de resposta

Resumo Sistemas embutidos correm varias aplicações potencialmente críticas e com restrições a varios niveis simultaneamente. Estas aplicações distribuidas correm sobre sistemas multiprocessador que deverão providenciar recursos suficientes de maneira a satisfazer as necessidades minimas de performance e largura de banda da aplicação. No caso de aplicacoes ditas "hard real-time" uma falha a esse nivel poderá ter consequencias desastrosas. Tais aplicações requerem portanto uma analise bastante conservadora. A automação de tal análise é um dos objectivos da ferramenta Heracles. Nesta tese apresentam-se varios metodos de análise, alguns deles originais, e demonstra-se a respectiva implementação na ferramenta Heracles. Há particular foco na descrição por grafos da aplicação distribuida, análise do debito da aplicação, escalonamento e mapeamento nos varios processadores do sistema, bem como alguns algoritmos de pós-optimização.

palavras-chave Scheduling, multi-processor system, embedded system, mapping, response time

Abstract Embedded systems often run several critical, time-constrained, applications simultaneously. These systems run atop multiprocessor systems-on-chip that must provide enough resourses such that the application's throughput constraints are satisfied. In the case of hard real-time applications any missed deadline might have disastrous results. Such applications therefore require a strictly conservative analysis. Automation of such analysis is the goal of the Heracles tool, originally created on NXP Semiconductors, and updated during the course of this thesis. This paper presents several analysis methods, some of them original, and their respective implementation on Heracles. It has focus on some of the possible dataflow graph depictions of the distributed application, throughput analysis, task scheduling and mapping, and some pos-optimization algorithms to compute slice times.

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem description . . . 2
- 1.3 The Heracles tool and the employed toolchain . . . 2
- 1.4 Objectives . . . 3
- 1.5 Achievements . . . 3

2 MultiProcessor Systems on Chip **5**
- 2.1 The Hijdra project . . . 6
- 2.2 Heracles MPSoC file definition and System implementation . . . 6

3 Algorithmic Complexity **9**
- 3.1 Big-O Notation . . . 9
- 3.2 Complexity Classes . . . 10
  - 3.2.1 Class P . . . 10
  - 3.2.2 Class NP . . . 10
- 3.3 P vs NP . . . 11
- 3.4 NP-Completeness . . . 11
- 3.5 Class NP-Hard . . . 11

4 Computation Models **13**
- 4.1 Definitions . . . 13
  - 4.1.1 Graph . . . 13
  - 4.1.2 Directed Graph . . . 13
  - 4.1.3 Paths and Cycles . . . 14
  - 4.1.4 Directed Acyclic Graph . . . 14
- 4.2 Dataflow Graphs . . . 14
- 4.3 Synchronous Dataflow (SDF) Graphs . . . 16
- 4.4 Homogenouos Synchronous Dataflow (HSDF) Graphs . . . 17
- 4.5 Cyclo-Static Dataflow (CSDF) Graphs . . . 18
- 4.6 Analytical properties of dataflow graphs . . . 19
- 4.7 CSDF Conversion to HSDF . . . 20
- 4.8 Deadlock detection . . . 21
- 4.9 Implementation of dataflow graphs on Heracles . . . 22
  - 4.9.1 Graph Datatype . . . 22
  - 4.9.2 CSDF Actor Datatype . . . 22

4.9.3 CSDF file definition . . . 23
4.9.4 Implementation of the CSDF to HSDF Conversion Algorithm . . . 24

5 Timing Analysis **27**
5.1 Maximum Cycle Mean and Throughtput . . . 27
5.2 Execution Time vs Response Time . . . 28
5.3 Worst Case Response Time Analysis . . . 29
5.4 TDM Response Model and Task Grouping . . . 31
5.5 Maximum Cycle Mean Algorithms . . . 31
5.5.1 Howard and Szymanski . . . 32

6 Symbolic Simulator **33**
6.1 State Space Analysis . . . 33
6.2 Throughput Revisited . . . 34
6.3 Implementation . . . 35
6.4 Buffer Size Calculation . . . 36

7 Scheduling and Mapping **37**
7.1 Scheduling Strategies . . . 37
7.1.1 Fully Static Schedules . . . 38
7.1.2 Self Timed Scheduling . . . 38
7.1.3 Dynamic Scheduling . . . 38
7.2 Multiprocessor Scheduling Complexity . . . 39
7.3 Scheduler Implementation . . . 39
7.3.1 Connection Model and Map tag . . . 43

8 Deadline Extensions **45**
8.1 Deadline Extension Pool . . . 45
8.2 Deadline Optimization . . . 46
8.3 Finding Slice Times Through Deadline Optimization . . . 46

9 Pos-optimization of slice times **49**
9.1 Binary Slice Allocator . . . 49
9.2 Random Slice Allocator . . . 50

10 Case Study **53**

11 Conclusions and Further Directions **59**

Glossary **62**

A Wlan receptor SDF file descriptor **63**

B TD-SCDMA file descriptor **67**

C MPSoC File descriptor **69**

List of Figures

1.1 A simple diagram for a version of Heracles prior to this work. 3
1.2 A simple diagram of the current Heracles version. 4

2.1 A simplified Multiprocessor System on Chip (MPSoC) diagram 6
2.2 Template tile for the Hijdra architecture. 7
2.3 A simple MPSoC description file. This is the system used for a WLan system 7
2.4 Ocaml processor and system data structures definition. 8

3.1 Venn diagram for the various complexity groups. 12

4.1 A simple graph.
V={A,B,C,D,E} E={(A,B),(A,C),(A,D),(C,B),(C,D),(C,E)} 13
4.2 A simple directed graph.
V={A,B,C,D,E,F} E={(A,B),(A,C),(B,D),(C,D),(D,E),(D,F),(E,A),(F,A)} . 14
4.3 A simple tree-like DAG 15
4.4 Comparison of dataflow models. 15
4.5 A SDF graph 16
4.6 An HSDF graph 18
4.7 A csdf graph 19
4.8 HSDF conversion from the CSDF on figure 4.7 21
4.9 The CSDF file descriptor for the CSDF graph presented in 4.7 24
4.10 Another CSDF graph. 24
4.11 The HSDF correspondig to figure 4.10 25

5.1 Task execution time 28
5.2 Task response time 29
5.3 A TDM processor. 29
5.4 A very simple HSDF graph. 30
5.5 Both tasks mapped and scheduled. 30
5.6 Actor separation to attain tighter throughput analysis. 31

6.1 A simple dataflow Graph 34
6.2 State Space exploration of the dataflow graph 6.1 34

7.1 Various scheduling policies. 38
7.2 A simple HSDF application graph. 40

7.3 The graph from 7.2 with a dependency added from scheduling. Notice that the execution order must now be A → B → C → D, due to the fact that all task dependencies must be fulfilled before a task might be enabled. 40
7.4 A repeated schedule. 40
7.5 Step 1 of the scheduling algorithm. On the right the graph's respective DAG. 41
7.6 Step 2 of the scheduling algorithm. 41
7.7 Step 3 of the scheduling algorithm. 42
7.8 Step 4 of the scheduling algorithm. 42
7.9 Final step of the scheduling algorithm. 42
7.10 A dataflow diagram for the scheduling algorithm. 43

8.1 A dataflow graph with its respective deadlines 46
8.2 A generic linear program to optimize sums of deadlines. 46
8.3 System of linear equations for the Linear slicer. 47

9.1 Flowchart for the binary slice allocator. 50
9.2 Flowchart for the random slice allocator. 51

10.1 A Wireless Lan 802.11a decoder 53
10.2 A TD-SCDMA job 54
10.3 Simulated WLan application. 57

List of Tables

5.1 Comparison of MCM algorithms 32

10.1 Processor use with scheduled WLan application. 55
10.2 Processor use with scheduled TD-SCDMA application. 56
10.3 Processor use and slice times on scheduled TD-SCDMA application after pos-optimization with binary slice allocator. 56
10.4 Processor use and slice times on scheduled WLan application after pos-optimization with random slice allocator. 56

Chapter 1

Introduction

1.1 Context

Heterogeneous multiprocessor systems-on-chip form the basis for current embedded systems. Such systems are present in advanced car audio systems, chips with wireless functionality, the digital signal processing systems present on our television sets, flight control systems and telecommunication systems to mention only a few places where they are currently being employed.

Various types of applications operate on those systems-on-chip. Their functionality varying at least as much as the different types of embedded systems.

In this thesis the focus is on hard real-time applications running on heterogeneous multiprocessor systems-on-chip. One of the most popular misconceptions regarding real-time systems is that they're all about speed and throughput. Although having fast and high throughput systems is a plus, the main concern of real-time computation is to make sure that the system respects all of the required constraints and meets all of the imposed deadlines under any condition. A real-time system must be predictable. That is, its functional and timing behavior should be as deterministic as necessary to satisfy system specifications.

As its usually the case with technology, one wants to get the most out of the system. That usually means running as many applications as possible, or add as much functionality as possible into a single application. On the other hand, we must respect all of the application's constraints such as latency and throughput that might be disrupted by the increased response times due to added functionality or other concurrent applications. To such an end, we must know the requirements of the application. The application must therefore be modeled and analyzed in a rigorous way. In the case of hard real-time applications, all of the analysis and approximations must be conservative, since a relaxed guess could lead to missed deadlines and the potentially disastrous consequences that such situation would bring.

Given that a formal analysis is required, it makes sense to automate it whenever possible. That is one of the goals of the Heracles tool, initially created at NXP Semiconductors and improved with extra functionality during the course of this thesis. The other one is to serve as testbed for new approaches and algorithms.

1.2 Problem description

Real-time computing is an ever growing market and a wide open research area with direct payoffs to current technology. So it is a surprise to notice that much of a real-time system design and implementation is still based on heuristics and empirical techniques that can be highly unpredictable.[17]

It was already mentioned that real-time systems must react within precisely bounded time constraints to events in the environment. As a result, the correct behavior of such systems depends not only on the value of the computation, but depends as well on whether that value was produced on time. A response that occurs too late can be useless or even downright dangerous, jeopardizing the entire application flow[21][17].

Still, the correct implementation of a real-time system is very complex. It depends on a multitude of factors:

- does the operative system in place offer real time guarantees?
- does the communication network of the chip offer real time and bandwidth guarantees?
- are execution times of the various tasks correctly measured?
- in what order should the distributed tasks execute?

Development costs for such complex applications can be quite high. To avoid further expenses due to a bad implementation all of the algorithms and applications should be thoroughly tested at all design stages. The presented tool, Heracles, seeks to help in that task by performing formal static analysis on a dataflow graph representation of the distributed application. This allows, among other things, detection of possible deadlocks. It is also possible to determine the application throughput and estimate buffer usage during design or prototyping stage. The tool can also devise a valid schedule (if at all possible) for an application given a set of throughput constraints.

Although a great deal more of testing and validation is required for a successful implementation of a real time system atop an heterogeneous multiprocessor system-on-chip, it can be appreciated the need for automated analysis tools such as Heracles.

1.3 The Heracles tool and the employed toolchain

The Heracles tool was created as part of Hijdra project at NXP Semiconductors. Its a static analysis tool, i.e. it does not need to execute any code from the application to analyse. Instead it relies on a graph representation of it to perform its analysis. One of Heracles' goals is to automate the resource allocation step, such as processor and slice time assignement, or the search for valid schedules for the application given throughput constraints. Another goal is to analyze the application behavior, to place bounds on buffer usage, return throughput information, based on worst case response times. All these topics will be discussed in detail later on.

The Heracles tool was itself programmed on the multi-paradigm language OCaml, mostly on a functional style. The language, due to its characteristics, strikes a very good balance between rapid development and maintainability as well as execution speed. In order to model some of the linear problems GLPK, the GNU linear programming toolkit, was used, as well as its OCaml bindings. To present the generated graphs in a graphical way the graphviz library is needed as the program can export to that format. All the presented results were obtained by using the Heracles tool, except when explicitly told otherwise.

Figure 1.1 depicts the original tool as it was before the performed changes on the context of this thesis.

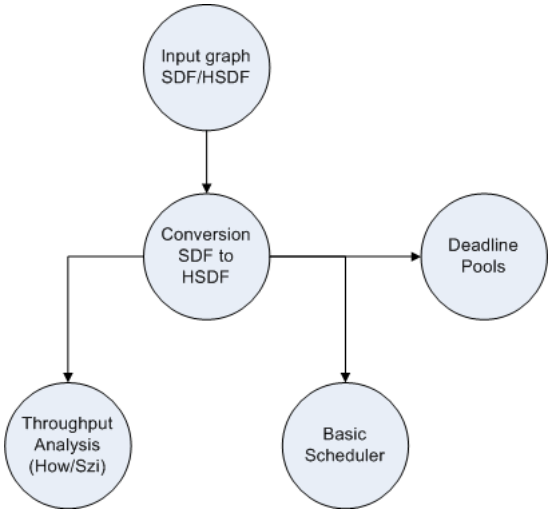

Figure 1.1: A simple diagram for a version of Heracles prior to this work.

1.4 Objectives

In practical terms, the first and main goal was to implement a way to work with cyclo static dataflow (CSDF) graphs within the Heracles tool. Following that, a way to work with response times instead of execution times was required so that we could perform worst case response time analysis as part of the graph analysis. Afterwards we've aimed at implementing a more streamlined work flow that would allows us to start with an application dataflow graph (being HSDF, SDF or CSDF) map it and schedule it to a predefined multiprocessor system, evaluate performance and perform some pos-scheduling slice time optimization.

As for the thesis text I hope to provide a general overview on the world of embedded systems as well as a description of the used methodologies and algorithms.

1.5 Achievements

The CSDF data structures and conversion algorithms were successfully implemented. SDF and HSDF graphs are now merely a restricted CSDF in order to maintain compatibility with minimum work to the user. A way to represent multiprocessor systems was created in order

to work with worst case response time models. Scheduling and mapping functionality was also implemented as well as two algorithms for the pos-optimization of slice times. A symbolic simulator for the dataflow graph was also created to calculate throughput and conservatively estimate buffer sizes.

Figure 1.2 outlines the final version of Heracles and provides a rough vision on the work flow the tool employs to analyse an application. The depicted algorithms will be detailed through subsequent chapters.

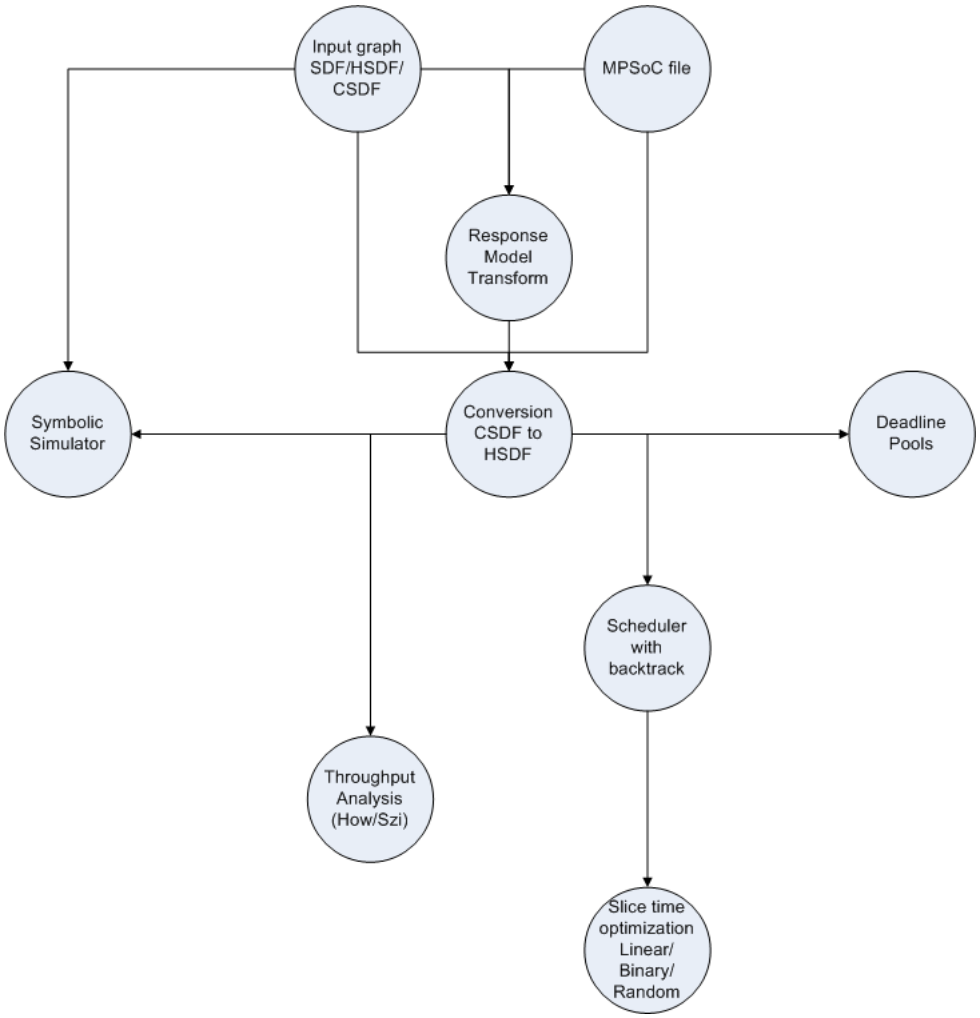

Figure 1.2: A simple diagram of the current Heracles version.

Chapter 2

MultiProcessor Systems on Chip

Multiprocessor Systems on Chip (MPSoC) are the latest incarnation of Very Large Scale Integration technology (VLSI). Firstly, a System on Chip is an integrated circuit that implements most or all of a complete electronic system. The system may contain memory, central processing units (CPU), specialized logic circuits (decoders, encoders), busses, and other digital functions.[2]

The architecture of the system is generally tailored to the application. We may find Systems-on-Chips on a great deal of applications, ranging from various consumer devices to industrial systems, cell phones, digital television sets and network routers. These applications do not use general purpose computer architectures because it is often not cost effective or simply because such architectures would not provide an adequate performance. This is the case of high-end networking and video devices. A general computer architecture, not having been built with that end in mind, will also not give much in the way of real-time guarantees. Such guarantees are very important to hard real-time systems.

An MPSoC is quite simply a System-on-Chip with multiple processing units. Also, very often, a MPSoC is composed by multiple heterogeneous processors each being a very different and specific processing unit such as a digital signal processor (DSP).Quite often MPSoCs contain as well one or more general purpose CPUs (ARM processors for instance). Memory in such processors might be heterogeneous as well. Devices may have embedded memory on-chip, single processors may contain local caches, memory might be shared and there is also the possibility of relying on off-chip commodity memory.

Considering the added complexity, why then use MPSoCs instead of a single processor or an homogeneous chip architectures? Quite simply, single general purpose processors or architectures with only one kind of processor might not provide enough performance for some applications. This is particularly true in the case of real-time video and communication systems in which it is imperative to keep up with the incoming data rates. Multiprocessors provide the necessary concurrency required to handle concurrent real-world events in real-time. High-end applications like video decoding must perform several tasks concurrently (discrete cosine transforms, Huffman decoding, and several more) and require true parallelism, not the apparent kind provided by a single processor, if the application is to meet the imposed

deadlines. Adding to those speed and performance requirements the fact that SoCs must be area and energy efficient and also provide proper IO connections, it is understandable the fact that industry is leaning towards heterogeneous MPSoC.[2]

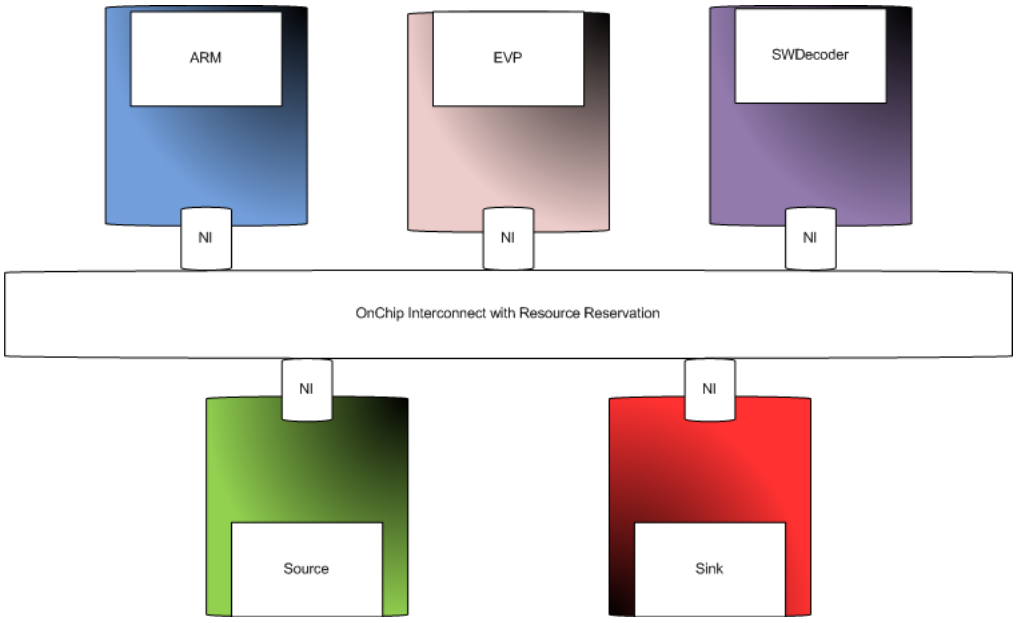

Figure 2.1: A simplified Multiprocessor System on Chip (MPSoC) diagram

2.1 The Hijdra project

The Hijdra project from NXP Semiconductors aims to develop an embedded heterogeneous multiprocessor system based on networks-on-chip that offers real-time guarantees and quality of service for its applications. By offering both timing and resource guarantees at the hardware level, tight estimations can be made for the jobs running on it.

The Hijdra MPSoC, consistis of a set of processing tiles connected by a network of routers. Every tile contains a single processor, its own local memory, and an interface to the network. The network is itself composed of routers of varied topology using unidirectional links.[3]

2.2 Heracles MPSoC file definition and System implementation

One of the key functionalities of the Heracles tool is to map and schedule applications to an arbitrary MPSoC and to analyze an application's behavior under such conditions. There is then a need to specify a depiction of the MPSoC to the tool. To that end a system description file containing the various parameters of interest must be given to Heracles to be subsequently parsed.

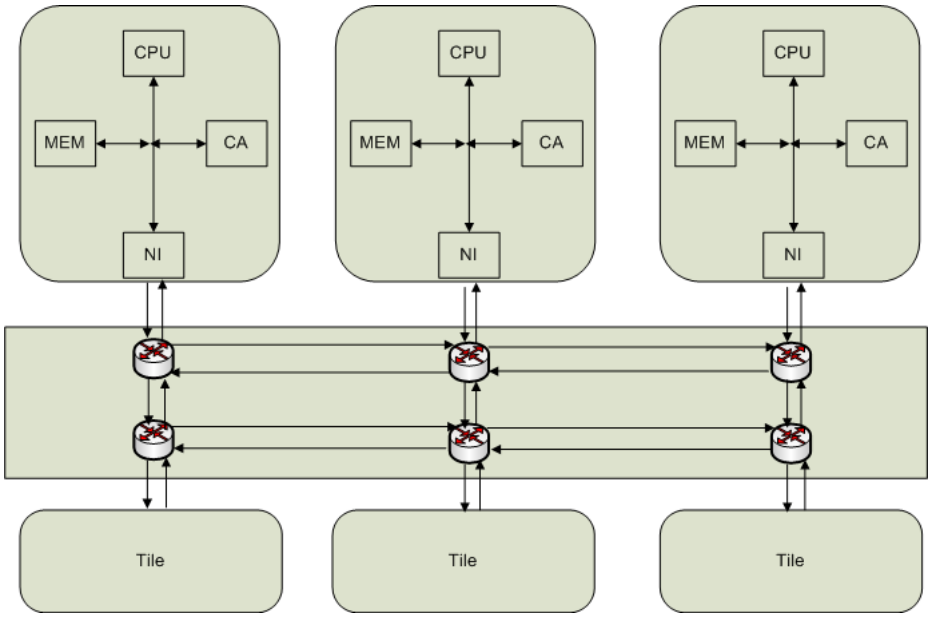

Figure 2.2: Template tile for the Hijdra architecture.

```
processor

name="EVP"      wheeltime=3500 type=1 sched="tdma" weight=100 (*usage=20*);
name="SwDecoder"    wheeltime=2000 type=2 sched="tdma" weight=100;
name="ARM"      wheeltime=3000 type=3 sched="tdma" weight=100;
name="Src"      wheeltime=2000 type=4 sched="tdma" weight=0;
name="Snk"      wheeltime=2000 type=5 sched="tdma" weight=0;
name="Snk2"     wheeltime=2000 type=6 sched="tdma" weight=0;
name="Snk3"     wheeltime=2000 type=7 sched="tdma" weight=0;

(*latency constrains*)
name="Lat1"     wheeltime=20000 type=8 sched="off" weight=0;
name="Lat2"     wheeltime=20000 type=9 sched="off" weight=0;

end
```

Figure 2.3: A simple MPSoC description file. This is the system used for a WLan system

We can see from figure 2.3 that a processor may have a name and type attributes, as well as several other properties. The name identifies the processor, and the type field defines which kind of tasks may be mapped onto it. That is, a processor of type x can only run tasks of type x. The field *sched* indicates the type of task scheduling that is performed on that processor. This allows us to consider the effects of scheduling on the execution time of a task. Currently the only supported type is time division (TDM) scheduling or no scheduling at all. That field will be discussed in more detail later on, on the timing analysis chapter as will the *wheeltime* field. Weight is a priority hint, useful for some of the implemented algorithms.

A simple parser was created with ocamllex and ocamlyacc to support the presented syntax. Those tools will, given the tokens and grammar, generate ocaml code containing a table driven parser. That parser will, when requested, return a system data type to the main application.

The system type represents an MPSoC architecture. Currently it is a simple list of processors. As such it doesn't yet model the intricacies of inter-processor communication or has a detailed scheme for memory accesses. Those aspects will have to be explicitly modeled on the dataflow graph should we wish to take them into account. The system is an encapsulated type and its actual definition is not known outside the module.

```
type schedtype = RoundRobin | TDMA | Off;;

(****** Processor Definition ******)

type processor = {
    proc_name: string;                      (* name of the processor*)
    proc_id: int;                           (*id of the processor *)
    mutable proc_wheel: int;                (*in tdma or round-robin *)
    mutable proc_ctxtime: int;              (*time it takes to context switch*)
    mutable proc_type: int;                 (*type of the processor*)
    mutable proc_scheduler_type: schedtype; (*how processor commutes between jobs*)
    mutable proc_weight: float;             (*priority hint for algorithms*)
    mutable proc_maxusage: float;           (*maximum usage of the proc per job*)
    mutable proc_slice_position: int;       (*define the position of a group slice*)
}

type system = processor list;;
```

Figure 2.4: Ocaml processor and system data structures definition.

Chapter 3

Algorithmic Complexity

During the course of this thesis, several algorithms, implemented on the Heracles tool, will be presented. Some of them are fast (howard, for instance) running in a few seconds time, others are quite intractable (like finding an optimal scheduling for an application) that can easily take days to finalize the computation. However such notions of speed are quite empirical. How can we formally say that an algorithm is faster than some other, and what should we expect from it as the number of elements to be computed increases? The answers to these questions lie in the field of complexity analysis. A brief introduction shall be presented here in order to allow for a better understanding of the presented algorithms.

The key idea of complexity analysis is to measure time and memory space as a function on the lenght of the input provided to an algorithm.

3.1 Big-O Notation

The exact running time of an algorithm is often a complex expression which depends on the particular computer where it is executed. Because of that we usually only estimate it. Yet, such estimation still provides a deep insight into the algorithm's behavior. A form of estimation, asymptotic analysis, seeks to understand the running time of an algorithm when run on large inputs on a machine independent way.[18]

From the analysis of an algorithm we devise a function of the input $f(n)$ that represents the running time of the algorithm for an input of size n. We consider only the higher order term and disregard any coefficient and lower order terms since that higher term will dominate the remaining terms on large inputs [1].

$$f(n) = 6n^3 + 2n^2 + n \tag{3.1}$$

Let us assume equation 3.1 is the running time expression of an algorithm obtained from analysis. Disregarding the coefficients and the lower order terms, we can say that $f(n)$ is asymptotically n^3. On big-O notation we write that relationship as $f(n) = O(n^3)$.

[1] That is, as the input approaches infinity, hence the name asymptotic analysis.

On a more formal way:

Let f and g be functions such that $f, g \mapsto R^+$. Then $f(n) = O(g(n))$ if, for the positive integers c, n_0 with $n > n_0$

$$f(n) \leq c \times O(g(n)). \tag{3.2}$$

When $f(n) = O(g(n))$ we can say that $g(n)$ is an asymptotic upper bound to $f(n)$.[18]

Considering equation 3.1 we verify that $f(n) = O(n^3)$ and even $f(n) = O(n^4)$. We cannot say $f(n) = O(n^2)$, though, since there is no value c that for $n > n_0$ verifies the equation 3.2.

3.2 Complexity Classes

3.2.1 Class P

The difference in growth between an exponential function and a polynomial one is quite large, to the point that we may consider that algorithms with polynomial time complexity, regardless of coefficients, to belong on the same class of algorithms. That class is called P. Formally, P is the class of algorithms that can be solved on polynomial time by a deterministic Turing machine[2]. Problems from this class are generally considered to be solvable, in a practical way, on a computer.

Examples of problems in P are the Shortest Path problem, solved by Dijkstra's Algorithm in $O(n^2)$ and the sorting algorithm BubbleSort, also of $O(n^2)$ complexity.

3.2.2 Class NP

Algorithms on the P class generally avoid a brute-force approach to solving a problem. However, in certain other types of problems there simply is no such luck, and currently there are no known polynomial algorithms to solve them. It may happen that these problems have polynomial time algorithms yet undiscovered, but, on the other hand, the possibility exists that simply there is no algorithm to solve them in polynomial time, and they are intrinsically difficult. An interesting point is that although the search for a solution is hard, to verify that solution often only polynomial time is required[18]. The group of such problems is called NP, and is defined as being the class of algorithms that have polynomial time verifiers[18]. An equivalent definition is to say that NP is the set of decision problems solved in polynomial time by a non deterministic Turing machine. The best method currently known to solve NP problems deterministically uses exponential time[19].

This is an important class for it contains many problems of practical interest such as the Hamiltonian Path problem and the Boolean Satisfiability problem.

[2] A mathematical abstraction of a computational machine.

3.3 P vs NP

As seen, solutions to the algorithms belonging to class NP can be quickly verified and solutions to algorithms in class P can be quickly computed. However this does not mean that $P \neq NP$, it only means that currently no algorithm is known to solve NP class problems in polynomial time. In fact, this particular question of whether $P = NP$ is one the greatest problems to solve on theoretical computer science and mathematics.

3.4 NP-Completeness

There is an important relation in the class of NP problems. It can be shown that there are certain types of problems in NP that can be reduced in polynomial time into another problem also in NP.[18] For instance a SAT problem can be converted in polynomial time into a traveling salesman problem. The class of those problems is called $NP - Complete$. Formally a problem C is in class NP if and only if:

1. C is in NP
2. Every problem in NP is polynomial time reducible to C

As such, if a solution in polynomial time can be found for a single problem in the $NP - Complete$ class, the same solution can be applied to every member of the class. Also this would imply that $P = NP$. Both the traveling salesman problem and the SAT problem are $NP - Complete$.

3.5 Class NP-Hard

There are more classes of problems besides P and NP. A problem, whether a member of NP or not, to which we can transform an NP-complete problem in polynomial time, will have the property that it cannot be solved in polynomial time unless $P = NP$. We might say that such a problem is "NP-hard", since it is, in a sense, at least as hard as the NP-complete problems. From the definition, all $NP - Complete$ problems also belong in the class $NP - Hard$. On figure 3.1 is shown how the different complexity groups relate amongst themselves.

As seen from figure 3.1 there are problems in $NP - Hard$ that are not in $NP - Complete$. One such problem is the halting problem. The halting problem can be stated has: "given an input program, will it run forever or come to an halt at some point?". For turing complete idioms that problem is undecidable. That has important repercussions on analysis and is the reason non turing complete dataflows are prefered for our purposes. We wish to be able to detect deadlocks (program halts) during static analysis.

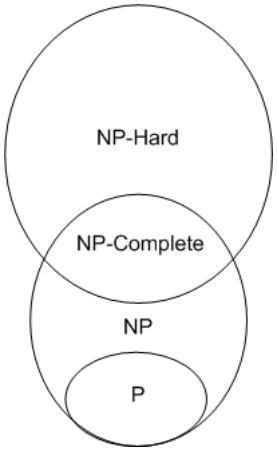

Figure 3.1: Venn diagram for the various complexity groups.

Chapter 4

Computation Models

This chapter deals with the way distributed applications can be modeled. There are, of course, several possible ways to represent a distributed application of which only a subset is presented. In Heracles, for reasons that will soon be explained, we've choosen to employ dataflow graphs. Dataflow graphs are, as the name implies, a certain type of graph, extended in notation to express some of the distributed application requirements like dependencies between iterations and execution times. A brief discussion of graphs ensues followed by a formalization of the dataflow graphs themselves.

4.1 Definitions

4.1.1 Graph

A graph is an ordered pair of disjoint sets (V,E) such that E is a subset of the set of unordered pairs of V. The set V is the set of vertices and E is the set of edges. An edge is denotated (A,B) and is said to join the vertices A and B. In a graph, the edge (A,B) means exactly the same edge as (B,A)[7].

Usually we do not think of a graph as a pair of sets, but as a collection of vertices joined by edges. The easiest way to describe a graph is perhaps to draw it as in figure 4.1.

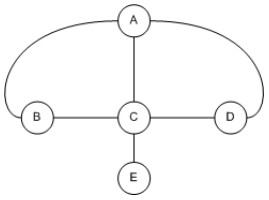

Figure 4.1: A simple graph.V={A,B,C,D,E} E={(A,B),(A,C),(A,D),(C,B),(C,D),(C,E)}

4.1.2 Directed Graph

If the edges are *ordered* pairs of vertices then we get the notion of a directed graph. In such graph the edge (A,B) is different than the edge (B,A). An ordered pair (A,B) is an edge directed from vertice A to vertice B. It can be said that the edge has a beginning at node A

and an ending at node B. We can also refer to the first vertice as the source and the second vertice as the sink. The dataflow graphs used to model an application are extended directed graphs where a directed edge (A,B) represents a communication channel.

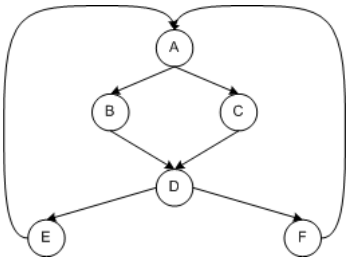

Figure 4.2: A simple directed graph.
V={A,B,C,D,E,F} E={(A,B),(A,C),(B,D),(C,D),(D,E),(D,F),(E,A),(F,A)}

4.1.3 Paths and Cycles

A walk is a finite non-null sequence $W = v_0e_1v_1e_2v_2...e_kv_k$ whose terms are alternately vertices and edges such that for $1 \leq i \leq k$ the ends of e_i are v_{i-1} and v_i. The integer k is the length of W.
In a simple graph a walk is determined simply by the sequence of its vertices. If the edges of W are distinct W is called a trail. In addition if the vertices of W are distinct W is called a path.
Two vertices (A,B) are said to be connected if exists a path from A to B.

A walk is closed if its initial and final vertex are the same. A closed trail whose origin and internal vertices are distinct is a cycle.

Cycles, and cycle detection are a very important part of the dataflow graph analysis. It will be seen on further chapters how an application throughput is related to cycles present of the dataflow graph of the application.

4.1.4 Directed Acyclic Graph

A directed acyclic graph (DAG) is quite simply a directed graph that contains no cycles. A spanning tree is a well known example of a DAG.
DAG's are quite an useful aid to schedule concurrent applications on multiprocessor systems as will be explained on the scheduling chapter.

4.2 Dataflow Graphs

As mentioned, there are several ways in witch we can model a distributed application. On the Heracles tool we've restricted ourselves to specific classes of so called dataflow graphs (HSDF, SDF, CSDF) for several reasons. First and foremost, unlike some other models (like Boolean dataflow and Petri Nets), these are not turing complete dataflow models, meaning these models are less expressive than a Turing machine. This, in turn, allows us to detect deadlocks in the application during the static analysis.

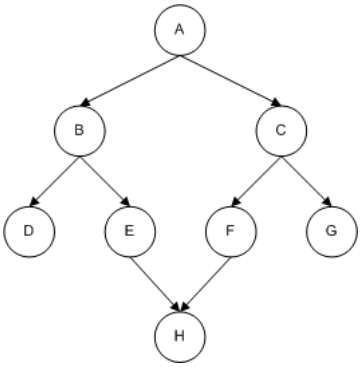

Figure 4.3: A simple tree-like DAG

Still, these models capture the essence of parallel task execution and allow for the calculation of the maximum attainable throughput and to establish bounds to buffer usage as well as to express latency and buffer constraints (although not directly)[4][5]. Another very important property of the used models is that they are fully deterministic models. This means that equivalent runs with the same inputs must yield the same results every time. This is important given the context of hard real-time application analysis since it is determinism that allows us to meet and provide guarantees for the application needs in terms of throughput and buffer size in the course of several or indefinite executions.

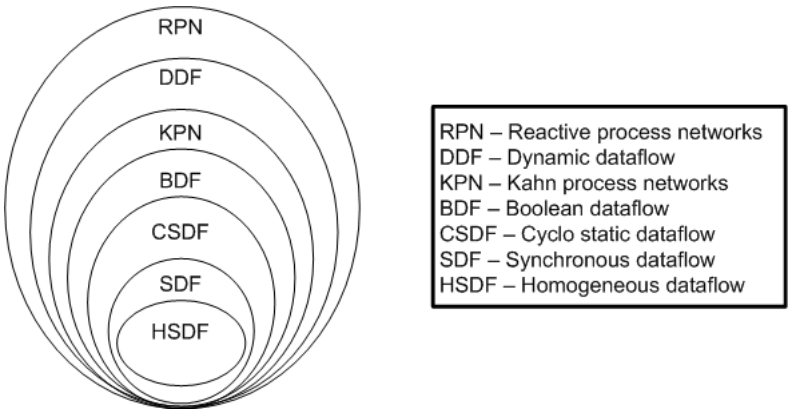

Figure 4.4: Comparison of dataflow models.

Figure 4.4 shows the relation amongst various dataflow representations. Boolean dataflow and larger are turing complete, and although more expressive dataflow types, because of that they no longer serve our purposes for static analysis.

On the studied dataflow graphs the vertices or nodes are called actors or tasks. They're responsible for transforming input data streams into output streams and as such we can view them as computations over provided data. On the lowest level an actor is an atomic part of the application or algorithm, with a well defined set of input and output data and a specific function (for instace a frame decoder task on a wireless LAN). Actors will ultimately

be mapped as running processes on some specified processor. Their behavior is generally described in a host language such as, e.g., C, LISP.

The edges between actors on the dataflow graph represent channels which carry streams of data from the output of one actor to the input of another. The atomic data object, carried by the edges, is called a token. In the context of our models these channels are considered to be unlimited first-in-first-out (FIFO) queues.

An actor fires or is enabled (that is, starts its execution), consuming a certain amount of time in order to perform its computation, whenever its firing rule is evaluated as true. This firing rule is a boolean function, different across the various types of dataflow graphs, and will be better defined on subsequent sections. The amount of time an actor takes to execute, when uninterrupted by the processor, since enabling until completion, is referred to as the actor's execution time. Furthermore, when an actor is executed it consumes a certain amount of tokens from its input channels and produces a number of tokens to its output channels.

4.3 Synchronous Dataflow (SDF) Graphs

This is the most commonly used dataflow graph. It was presented by Lee on [5]. This restricted dataflow model poses restrictions on the firing of actors: the number of tokens produced (consumed) on each output (input) is a fixed number known at compile time. The number of tokens produced or consumed by each SDF actor are represented in illustrations of SDF graphs as numbers near the edge source and sink respectively. As mentioned, these edges represent FIFO buffers in physical memory.

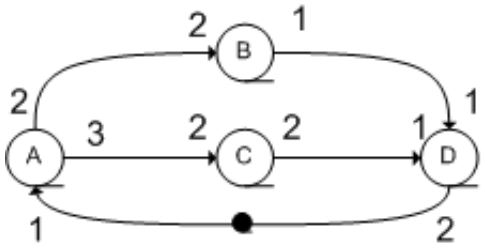

Figure 4.5: A SDF graph

The edges in a SDF graph may contain initial tokens, also known as delays. Edges with delays may be interpreted as data dependencies across iterations of the graph. Delays are represented graphically as bullets (•) on the edges of the graph. We indicate more than one delay on the edge by a number alongside the bullet.

We can formally define SDF graphs as a tuple (V, E, δ, T, P, C) where:

- V is the set of actors
- $E \subseteq V \times V$ is the set of directed edges

- $\delta : E \rightarrow \mathbb{N}$ is a function that given a directed edge $(u, v) \in E$, returns the initial number of tokens for that edge
- $t : V \rightarrow \mathbb{R}^+$ is a function that given an actor returns its worst case execution (or response) time of an actor $v \in V$
- $P : E \rightarrow \mathbb{N}$ is a function that given a directed edge $(u, v) \in E$, returns the number of tokens produced on that edge by its source actor u.
- $C : E \rightarrow \mathbb{N}$ is a function that given a directed edge $(u, v) \in E$, returns the number of tokens consumed by its destination actor v

An SDF actor fires only if there are as many tokens on its input edges as required by the consuption function of all input edges. Accordingly, its firing function evaluates to true if, and only if, all of the actor's input edges have an equal or superior number of tokens than specified by the consumption value of those same edges.

A drawback to the SDF model is that assumes that each task behaves identically every time it is executed which might not always happen. As we'll see the Cyclo-static dataflow model handles that situation to a certain extent.

In figure 4.5 actor B can fire as soon as it has 2 tokens on its single input edge. From the graph we see that A produces 2 tokens as soon as it finishes execution. We may then conclude that B can fire as soon as A finishes its execution. The C actor is on the same conditions as B and may also be enabled as soon as A concludes its execution. So, it may execute its computation in parallel with B if running on a different processor.

As for a real-world hard real-time application we present an example (see figure 10.1) provided by NXP Semiconductors: an 802.11a Wireless Lan receiver modeled according to the SDF semantics.

4.4 Homogenouos Synchronous Dataflow (HSDF) Graphs

Both SDF graphs and Cyclo-static dataflow (CSDF) graphs can be transformed into equivalent HSDF graphs. This is required for the purposes of throughput analysis and scheduling. An HSDF graph is an even more restricted form of SDF graphs: all consumptions and productions are limited to a single token. A more detailed depiction of the analysis methods applied on HSDF graphs will be found on subsequent chapters.

As noted from figure 4.6 productions and consuptions aren't represented on the graph. They are all unitary so may remain implicit. An actor is fireable if there is at least one token in every one of its incoming edges and only in such case does the firing function returns true.

The SDF to HSDF conversion algorithm previously present on Heracles was replaced by the CSDF to HSDF conversion algorithm and as such will not be discussed here. A description of that algorithm can be found on [4].

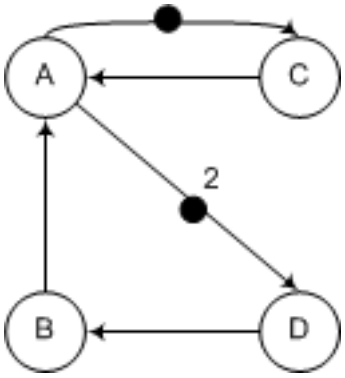

Figure 4.6: An HSDF graph

A parallel application using the HSDF semantics, provided by NXP Semiconductors is the TD-SCDMA of figure 10.2.

4.5 Cyclo-Static Dataflow (CSDF) Graphs

Cyclo-Static dataflow is a relatively new model for the specification of distributed applications. It is originally presented on [6].

The CDSF paradigm is a deterministic and non Turing complete extension to SDF that still allows for static scheduling and, thus, a very efficient implementation of an application. In comparison with SDF it is more versatile for it also supports algorithms with a cyclically changing, but predefined, behavior. This capacity may result in a higher degree of parallelism and, hence, a higher throughput, shorter delays and a lesser buffer memory use.[6]

In cyclo-static dataflow graphs each actor $v_j \in V$ has an execution sequence $[f_j(1), f_j(2), ..., f_j(P_j)]$ of length P_j. The meaning of this sequence is as follows: each firing of task v_j executes code from function $f_j((n-1) \mod P_j+1)$ with an execution time of $t(j)$. As a consequence, productions, consuptions and execution times are also sequences in CSDF graphs. The production of v_j on edge e_u is represented as a sequence of constant integers $[p_j^u(1), p_j^u(2), ..., p_j^u(P_j)]$. The n_{th} time actor v_j is executed produces $p_j^u(n-1 \mod P_j+1)$ tokens on edge e_u. The consuption of the actor v_k on the edge e_u is completly analogous and will be represented by the function $c_k^u(n)$. The n_{th} consuption of an actor v_k on the edge e_u will similarly be $c_k^u(n-1 \mod P_k+1)$. As an example, let's suppose there exists a cyclo-static actor with a production sequence on one of its edges of $4, 5, 6$. Such actor will produce 4 tokens on its first firing, 5 tokens on its second firing and 6 tokens on its third firing. On its fourth firing it will again produce 4 tokens and the sequence repeats itself for as many times as the actor is enabled. Should $P_j = 1$ for all of the actors belonging to the CSDF graph then it will degenerate into the already known SDF graphs.

The firing rule for a CSDF graph evaluates as *true* for its n_{th} firing if and only if all input edges contain at least $c_k^u(n-1 \mod P_k+1)$ tokens. The actor A in figure 4.7 produces one token on its first execution. On its second and third execution it produces nothing and on its fourth execution produces again one token.

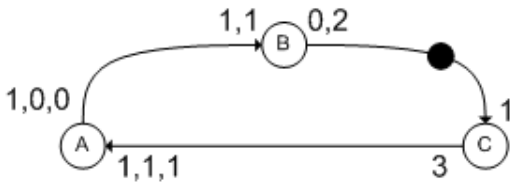

Figure 4.7: A csdf graph

4.6 Analytical properties of dataflow graphs

The structure of presented dataflow graphs can be compactly represented by its topology matrix, Γ. This matrix contains one column for each of the graph's actors and a row for each edge. On both SDF and HSDF the entry in $\Gamma_{(i,j)}$ corresponds to the number of tokens produced by the actor j on edge i. However, if instead of producing, the actor consumes tokens on edge i that entry is negative. If an actor neither produces nor consumes tokens on that edge i $\Gamma_{(i,j)} = 0$. As for the CSDF graph the same rules apply with the only difference being that for the pair i, j we must consider the sum of all productions or consumptions on that edge. The topology matrix for the graph in figure 4.7 is the following:

$$\Gamma = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 2 & -1 \\ -3 & 0 & 3 \end{bmatrix}$$

An essencial step in the implementation of a CSDF graph on a multiprocessor system is the static scheduling of the actors. The divised schedule will be indefinitly repeated on the various processors. For a proper run-time execution, all data must be available when a task is executed and the amount of data on the buffers must remain non-negative and bounded. If these conditions are satisfied the graph is said to be consistent.

We may prove the consistency of a graph once the balance equation 4.1 is solved, thus obtaining the repetition vector, designated q, and proving a nullspace exists for the equation system. A dataflow graph is consistent, and therefore has consistent and bounded sample rates, if q exists [6][5]. In order to obtain the HSDF equivalent of the CSDF graph we must also solve the balance equations so we may say that if an equivalent HSDF exists, the original CSDF is consistent. Furthermore if a deadlock free static scheduling can be found, the graph is said to be *live*[6].

The repetitions vector q is extracted from the topology matrix. For a graph with n actors q is a column vector of lenght n with the property that if each actor i is enabled q_i times the number of tokens on each edge remains unchanged. Since that property holds for every $c \cdot q$ where c is an integer, q is considered to be the smallest integer vector where that property holds. Having calculated q we can then proceed to generate infinite schedules for dataflow graphs while maintaing bounded and consistent buffer sizes amonst actors[4]. The vector q is calculated by solving:

$$\Gamma \cdot q = 0 \tag{4.1}$$

A more detailed analysis on why this is happens, can be found on [5] regarding SDF graphs and on [6] for CSDF graphs.

4.7 CSDF Conversion to HSDF

In order to analyze a CSDF graph's throughput and schedule it (both issues will be analysed on subsequent chapters), we must first perform a conversion to its equivalent HSDF. The resulting equivalent HSDF graph has as many actors as specified by the original dataflow graph repetitions vector q and each of these actors represents an execution of the original actor.

Let G be a CSDF graph. The CSDF to HSDF conversion algorithm is as follows:

1. Construct the equivalent HSDF actors:
   For every task v_j of G construct q_j task instances: $v'_j(1), v'_j(2), ..., v'_j(q_j)$. These tasks will be the actors of the equivalente HSDF graph.

2. Add sequence-edges to model the in-order execution of G.
   For all tasks v_j of G:

   - For all $i_j : (1 \leq i_j \leq q_j) \rightarrow$ Construct an edge from $v'_j(i_j)$ to $v'_j(i_j + 1)$
   - Construct an edge from $v'_j(q_j)$ to $v'_j(1)$ with one initial token.

3. Add edges to model communication.
   For every edge e_u in G:

   - Determine the first invocation of task j $v_j(s^u_j(1))$ that produces data on edge e_u.

   - Determine $v_k(n^f_{k,j}(s^u_j(1)))$, i.e. the first invocation of task v_k that consumes tokens produced by $v_j(s^u_j(1))$. Recall that $\delta(u)$ returns the number of initial tokens on an edge and P_k is the period of actor k.

     $n^f_{k,j}(s^u_j(1)) = (\delta(u) \div c^u_k(P_k)) \times P_k + n^*_k$, with:
     $1 \leq n^*_k \leq P_k$
     $c^u_k(n^*_k - 1) \leq \delta(u) \mod c^u_k(P_k) < c^u_k(n^*_k)$

     In the equivalent HSDF graph, this corresponds to instance $i_k = n^*_k \mod q_k$ of v_k.

   - Determine the number of tokens $n_o(v_k)$ required to make invocation $v_k(n^f_{k,j})$ of task v_k executable:
     $n_o(v_k) = c^u_k(n^f_{k,j} \mod {}_1P_k) - (\delta(u) - c^u_k(n^f_{k,j} - 1))$.

     where $n \mod {}_1m = (n - 1) \mod m + 1$
   - Add edges to Homogenouos graph using the following algorithm:

     $n_k = n^f_{k,j}$;
     for $(i_j = s^u_j(1) \rightarrow q_j)\{$

$n_o(v_j) = p_j^u(i_j \mod P_j)$ while $(n_o(v_j) \neq 0)$ {
amount(i_j, i_k)=min$(n_0(v_j), n_o(v_k))$;
add an edge from $v'_j(i_j)$ to $v'_k(i_k)$ with $(n_k\ div_1\ q_k - 1)$ initial tokens.
$n_o(v_k) -= amount(i_j, i_k)$;
$n_o(v_j) -= amount(i_j, i_k)$;
while $(n_o(v_k) == 0)$\{

$n_k++; i_k = n_k \mod {}_1q_k$;
$n_o(v_k) = c_k^u(n_k \mod {}_1P_k)$;
}
}
} where $n\ div_1\ m = (n-1) div\ m + 1$

The presented algorithm and its demonstration can be found on [6].

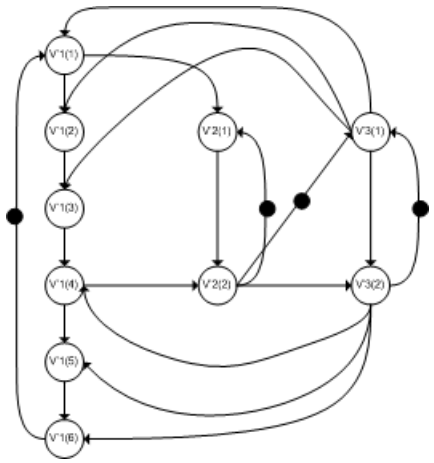

Figure 4.8: HSDF conversion from the CSDF on figure 4.7

4.8 Deadlock detection

If, during the course of an application execution, its not possible to enable any actor, that application is deadlocked. It can easily be understood that a deadlock condition is highly undesirable, particulary in hard real-time systems where there are strict deadlines to follow and system failure can have drastic consequences. One of the advantages of using one of the presented dataflow graphs for modeling distributed applications is that deadlocks can be detected during the design stage and avoided. In that context, deadlocks are caused either by inconsistency or by an insuficient number of tokens on one of the cycles of a graph. We've already seen how to detect buffer inconsistency. In order to detect whether insuficient tokens are a problem we can simply search for cycles with no delays on the equivalent HSDF. If such a cycle exists, at some point during execution, no actor may fire and we have a deadlock[4][5]. There are several efficient (i.e of polynomial complexity) algorithms for cycle detection[11]. Some of them are implemented in Heracles and will be discussed in more detail.

It can be seen that a deadlock exists on the presented CSDF 4.7. To do so we must analyze its equivalent HSDF (figure 4.8). A cycle with no delay can be found transversing actors $V_1(4) \rightarrow V_2(2) \rightarrow V_3(2) \rightarrow V_1(4)$.

4.9 Implementation of dataflow graphs on Heracles

4.9.1 Graph Datatype

The implementation of the graph data type on Heracles was already implemented by previous developers. It's definition is as follows:

type ('a, 'b) graph = 'a list * (('a * 'a) * 'b) list

This is a parametrized type having two parameters ('a can be replaced by any other type, such as a CSDF actor type and 'b by a custom edge type). The data structure is a tuple of which the first element consists of a list of type 'a (for our intents we can consider it a list of nodes) and having as second element another list representing the existing edges. Each element on that list is another tuple containing the source and destination nodes and an edge information structure 'b.

The graph datatype is written on funtional style and the dataflow graphs datatypes are layered on top of it. This definition however is not without problems, for should we require to search for a particular node based on information presented on a defined 'a node, a full linear search must be performed (an $O(n)$ algorithm). Also it is not trivial to find the adjecent nodes. It is easy, since there is a function for it, but it also implies a $O(n)$ search. For algorithms needing speed (such as the howard algorithm and the symbolic simulator), auxiliary custom data structures are created.

4.9.2 CSDF Actor Datatype

As mentioned, a CSDF datatype is built on top of the Graph datatype. Not only that, but this datatype also generalizes the previously implemented HSDF and SDF graphs. We are allowed to do so since it an HSDF graph can be understood as a special CSDF with a single period and unitary production and consumption of tokens. Likewise, an SDF graph is a CSDF restricted to a single period.

type csdf = (actor, arc_info) Graph.graph

We can see that the CSDF datatype is a Graph with 'a=actor and 'b=arc_info. Both actor and arc_info are records containig varied information, intrinsic to the dataflow, and some more fields required for some algorithms.

```
type actor = {
    ac_name: string;                        (*a unique name*)
    ac_id: int;                             (*a unique id*)
    mutable ac_exec_time_list: int list;    (* cyclic execution times *)
    mutable ac_response_time : int;         (* response time of actor *)
    mutable ac_index: int;                  (* for some algs it's handy to index actors*)
```

```
    mutable ac_orig: actor option;          (* in hsdf, from whom this copy is derived*)
    mutable ac_copy_nr: int;                (* in hsdf, nr of this copy, for sdf -1*)
    mutable ac_repetitions: int;            (* store actor repetions, see Topmat*)
    mutable ac_proctype: int;               (* processor type*)
    mutable ac_comaptag: int option;        (* comappimg actors, = tag -> = processor *)
    mutable ac_cluster_members: actor list; (* for cluster nodes *)
    mutable ac_slice: int;                  (* slice time*)
    mutable ac_group: int option;           (* group id None | Some of int *)
    mutable ac_map: string option;          (* processor to be mapped *)
    mutable ac_type: actype;                (* actor type*)
}

type arc_info = {
    mutable ar_delay: int;          (* delays present on the edge *)
    mutable ar_prod:int;            (* rate of production of tokens *)
    mutable ar_prod_list:int list;  (* production list for csdf *)
    mutable ar_cons:int;            (* rate of consumption of tokens *)
    mutable ar_cons_list:int list;  (* consumption list for csdf *)
    mutable ar_token_sz: int;       (* size of tokens *)
    mutable ar_buffer_sz: int;      (* for buffer size, eventually... TODO *)
    mutable ar_internal: bool;      (* flag for internal edge *)
    mutable ar_opt_channel_model:   (arc_info chfun) option; (*comm model*)
}
```

Presented above are the main data structures (mutable records) used in a CSDF dataflow graph. With this structures in place we have fully defined the dataflow graph. Although most record fields in both actor and arc_info are not touched by most algorithms they are nontheless declared mutable to facilitate creation and to be future proof. The datatypes defined here are encapsulated, and their actual definition is hidden from other modules. All access is performed through functions defined on the CSDF module. This will allow for a smooth migration to some other datatype should the need arise (as it did during the conversion from SDF to CSDF).

4.9.3 CSDF file definition

Heracles needs to know how the distributed application to be analysed is defined.
Following the same strategy as for the system file, a parser and lexer were created with ocamllex and ocamlyacc. The parser reads the file and outputs to the application a fully constructed CSDF graph. To define the application we merely define the csdf graph with all its attributes. This is a two stage process. Firstly we declare the actors and respective attributes. Following that, we declare the edges and corresponding properties. If undefined actors are detected, or any other sort of error is found (such as, for instance, production/consumption period mismatch between edges and actors), the program will exit outputting the discovered error and respective line where it was detected. A simple CSDF file descriptor example for the CSDF graph presented in figure 4.7 is presented here (figure:4.9). For a real world application, a Wireless Lan receiver (figure10.1), the graph file descriptor can be consulted in

the Appendix.

```
actors

name="A"    exec=10,20,10    slice=25 proct=1 group=1 map="P1";
name="B"    exec=20,30       slice=25 proct=1 group=2 map="P2";
name="C"    exec=30          slice=25 proct=1 group=3 map="P3";

arcs

src="A"    dst="B"    delay=0 prod=1,0,0 cons=1,1 tokensize=1;
src="B"    dst="C"    delay=1 prod=0,2 cons=1     tokensize=1;
src="C"    dst="A"    delay=0 prod=3 cons=1,1,1   tokensize=1;

end
```

Figure 4.9: The CSDF file descriptor for the CSDF graph presented in 4.7

A CSDF actor is identified by a name (also by a unique integer id that is automatically given by the application and is different even for copies of the graph). The execution time can be an integer, or a list of integers should the actor have a non unitary period. There is no specification of time units, so all actors implicitly share the same units. Apart from the name and execution time(s), all other tokens are optional and of use only for particular algorithms. The slice item stands for allocated slice time and is required for the response time calculation (more on that on the following chapter). The group tag is usefull for response time calculation and scheduling. Both proct and map are hints to the scheduler. Proct indicates the processor type on which the task is allowed to run (an MPSoC has the possibility of having more than one processor of the same type) and the map field explicitly maps a task to a specific processor.

As for the edge portion of the file its mandatory fields are the source and destination, respectively src and dst. Each of those attributes should have the name of an already defined actor lest the application complain.

All of the non mandatory fields have sensible default values should they be left undefined

4.9.4 Implementation of the CSDF to HSDF Conversion Algorithm

The conversion algorithm was implemented as described by the presented algorithm. For the presented graph 4.7 it returns the correct equivalent HSDF graph as depicted on 4.8. We present another example:

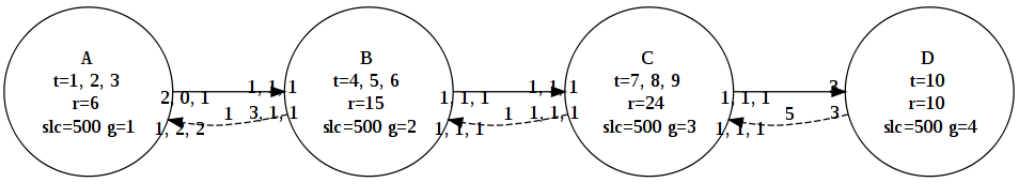

Figure 4.10: Another CSDF graph.

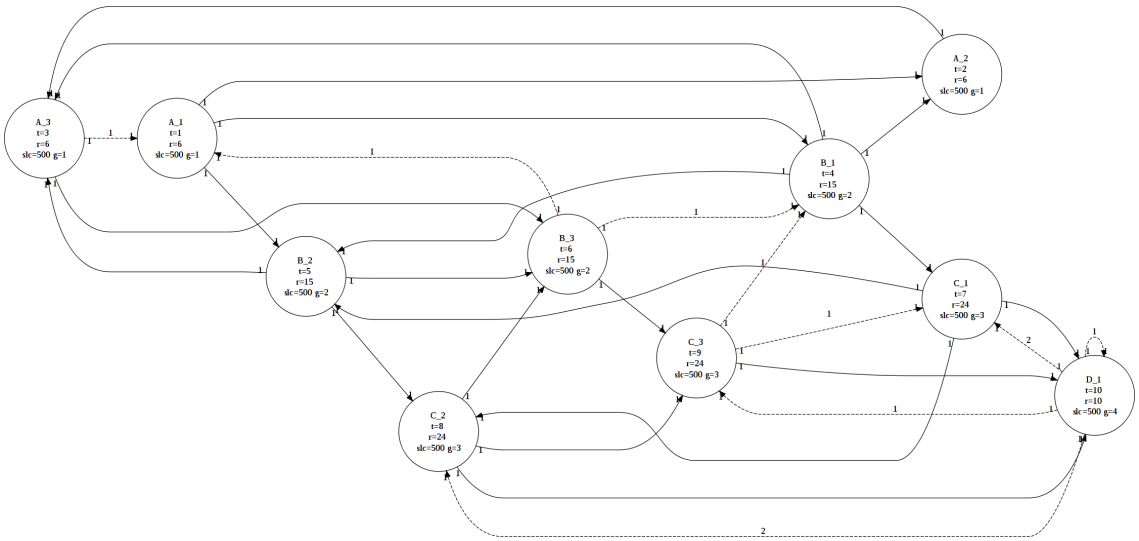

Figure 4.11: The HSDF correspondig to figure 4.10

Both these figures have been drawn automatically by the Heracles dot file exporter. It can be noticed that even for a seemingly simple CSDF graph its equivalent HSDF graph can be quite daunting.

Since an SDF graph is a particular type of CSDF the previous conversion algorithm (converting SDF's to HSDF's) in place on the Heracles tool was replaced by this new implementation.

A simple test script written in bash was created to make sure that the CSDF to HSDF conversion would yield the same results as the SDF to HSDF conversion on a multitude of different graphs, and further increase confidence on the new code. Specifically the script matched the graph throughput of both the old and new versions of Heracles. Throughput will be discussed in more detail in the following section.

Chapter 5

Timing Analysis

The whole point in having an analytical model for the representation of an application is to have a way in which we can give guaranties regarding whether we can satisfy the application temporal constraints. Our focus is on hard real time applications, and on such applications, the deadlines of tasks cannot be missed. That is to say, we must be able to insure that a minimum throughput is met, depending on the applications specifics. Also we may have the need to assert that some latency constraints are satisfied. We will not concern with latency constraints here, but more information, and a way to model latency constraints as throughput constraints on SDF and HSDF graphs can be found on [9].

5.1 Maximum Cycle Mean and Throughtput

The relation between a SDF/CSDF and HSDF was already established by means of the repetition vector q_g, obtained through the balance equations. The throughput analysis for a SDF or CSDF graph is performed on the respective equivalent HSDF graph. This is due to the fact that we can easily obtain the throughput of an HSDF and such throughput is identical to the one on the original graph[9].
First, let us define the Cycle Mean μ_c of a cycle c in an HSDF graph:

$$\mu_c = \frac{\sum_{i \in N(c)} t_i}{\sum_{e \in E(c)} \delta(e)}, \tag{5.1}$$

where $N(c)$ is the set of all nodes traversed by the cycle c, and $E(c)$ is the set of all edges traversed by the same cycle. The function t_i gives us the execution time of the task i and $\delta(e)$ returns the delays on edge e. We can now define the Maximum Cycle Mean (MCM), as beeing the larger of all μ_c.

$$\mu_g = \max_{c \in C(G)} \frac{\sum_{i \in N(c)} t_i}{\sum_{e \in E(c)} \delta(e)}, \tag{5.2}$$

where $C(G)$ is the set of all simple cycles present on graph G[4]. Tasks transversed on the cycle with the highest cycle mean are said to belong to the critical cycle.

The Maximum Cycle Mean of a dataflow graph is closely related to the maximum achievable throughput.
When, in a dataflow graph, actors are enabled as soon as there are tokens available, it is called a self-timed execution. The maximum attainable throughput of a graph $G = (V, E)$ is that of its self-timed execution, as no actor $v \in V$ can start firing without having enough tokens on all of its input channels, and any delay in the firing of an actor is of no use in increasing the number of firings of v itself or any other actor in the graph. The maximum possible throughput for an application is the one attained by its self-timed execution. That is, the throughput we get should all tasks fire up as soon as enough tokens are present on its input edges. This assumes, of course, enough resources are present to satisfy such demands. That said, the self-timed throughput (and therefore the maximum achievable throughput for the application is simply the inverse of the MCM.

$$T = \frac{1}{\mu_g}. \tag{5.3}$$

There is a relatively large set of polynomial complexity algorithms divised to calculate the MCM, and by consequence, the throughput of an HSDF graph. A comparison of those algorithms can be found on [11]. The Heracles tool implements two of those algorithms: Howard and Szimansky. They operate on the equivalent HSDF graph.

5.2 Execution Time vs Response Time

The execution time, as seen on previous chapters, is the amount of time a task takes to process its data, when executing stand-alone on the processor. However, due to budget constraints or other various considerations, very often we don't have a single application running on the MPSoC. As such, we must multiplex the resources amongst all applications, and cope with the throughput penaltys such sharing imposes. What this means is that usually a task takes longer to complete than specified by the execution time due to context switching and resource division. We call response time (r_t) to the time elapsed since the enabling (or firing) of the task until its completion when subjected to interruptions, context switches, preemptions and other effects due to resource arbitration. In a simple model, like a processor TDM scheduler, a task response time is a function of its execution time, the time slice given to the task and the turn time (or time wheel period) of the processor. Since task and application preemption is, for all practical purposes, non-deterministic and therefore very hard to model or analyze it is assumed disabled in the context of this work.

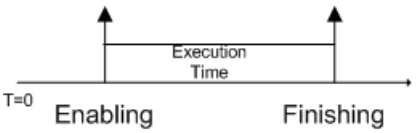

Figure 5.1: Task execution time

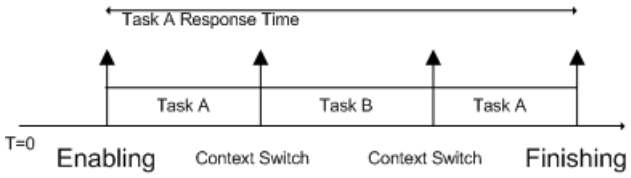

Figure 5.2: Task response time

5.3 Worst Case Response Time Analysis

The analysis performed by the Heracles tool used to use only execution times to compute the application throughput. This is acceptable only should all tasks run uninterrupted on the designated multi-processor system. As resources are expensive, this is often not the case, and several applications run on the same MPSoc.

The Heracles tool was updated to cope with response times. We do this by allowing the user to specify the target architecture configuration on a so called system file, fed to Heracles. The response times are calculated taking into account the system parameters and the execution times of the various tasks. The throughput of the application running can afterwards be computed by replacing the task execution times with the respective conservative response time, maintaining the throughput analysis algorithms unaltered.

Currently, in Heracles, the only supported type for intra-processor scheduling is a time division (TDM) model on which a processor is given a time wheel and a time slice is allocated to each task. No preemption is allowed since we cannot predict when it happens and we cannot, accordingly, offer any guarantees that the calculated response time is indeed conservative.

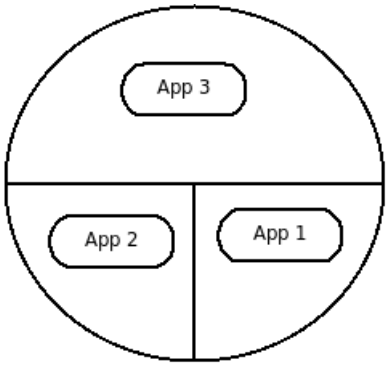

Figure 5.3: A TDM processor.

The formula for calculation of response times of a task in a TDM environment is given by:[1]

$$r_t(a) = (P - S) \cdot \lceil \frac{t(a)}{S} \rceil + t(a) \tag{5.4}$$

where P is the period of the processor time wheel, S is the slice allocated to the task in question and $t(a)$ is the execution time of the task should it be given the full use of the processor time. The equation 5.4 arises from the need to be conservative. So it is assumed

that, when enough tokens arrive to the task's input edges so that it may be enabled, the task itself will find the processor wheel on the possible worst position, i.e. right after its slice has ended. The remaining factors account for the fact that the task might not finish execution in a single slice.

Let us assume the existence of a simple application whoose dataflow graph is as follows:

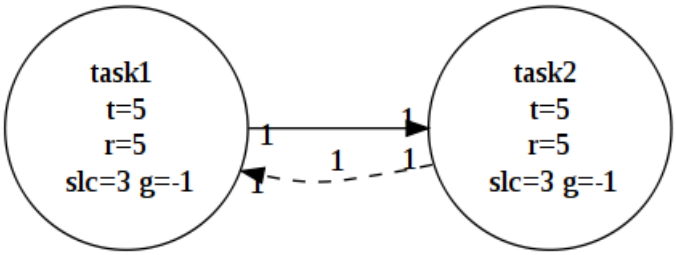

Figure 5.4: A very simple HSDF graph.

The MCM for the graph in figure 5.4, when running with full processor use, is of 10 time units. Let us now schedule both tasks to run each on a different processor, both processors with a time wheel period of 10 time units. To each task has been attributed a slice of 3 time units. The calculated response time of each task is 19 time units and the MCM has now risen accordingly towards 38 time units.

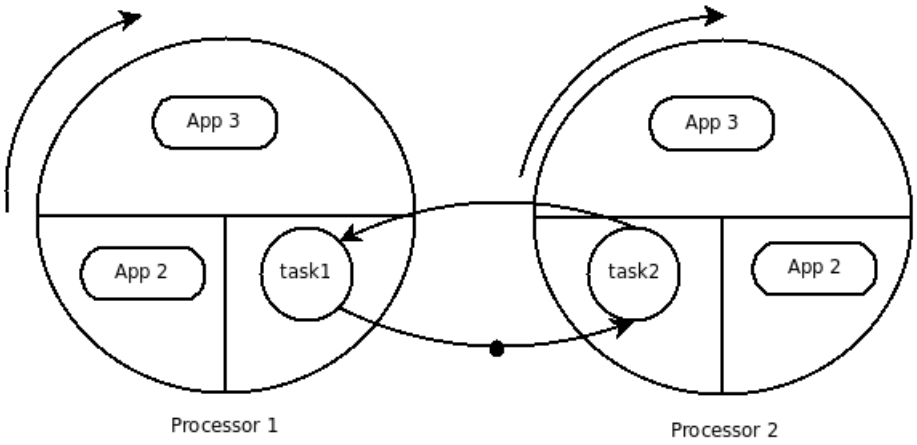

Figure 5.5: Both tasks mapped and scheduled.

Regarding the implementation, to replace execution times with response times it is a simple matter of running through each task and applying the computation. This is a very simple polynomial time algorithm.

5.4 TDM Response Model and Task Grouping

The values calculated with simple worst case analysis are quite pessimistic (as they must be in order for the analysis to remain conservative). Let us now assume that instead of assigning a slice to a single actor, we attribute a slice to a set of actors that we know to be (or force through scheduling) mutually exclusive (i.e. they are never executing concurrently) and we group those actors inside such slice. With such a strategy we can indeed lower our bounds for the worst case response time[1].

So if we have mutually exclusive actors running on the same slice S on a processor with a time wheel period P we can use equation 5.4 on each actor to model the response time of each actor and achieve the same results we would should each actor had a separated slice S. However situacions arise in which this approach is too conservative. For instance if an actor does not have to wait on input from a task running on another processor (all of its inputs are local) it could start its execution immediatly, not having to wait for the worst case scenario where it would be enabled just after its time wheel had elapsed. In such situation we can subtract the term $(P-S)$ from its response time equation. We can go further by noticing that there is never a wait caused by scheduling time between actors running on the same processor. We can therefore separate equation 5.4 in two terms, one accounting for the scheduling delay $r_s(a) = (P - S)$ and another accounting for the response time whithout the penalty attained for considering the time wheel on the worst position $r_x(a) = (P - S) \cdot (\lceil \frac{t(a)}{S} \rceil - 1) + t(a)$.

An actor on the original dataflow graph must now be represented by two actors. One a_s with an execution time of $r_s(a)$ receiving inputs from tasks running on other processors and forwarding them to the second actor with an execution time $r_x(a)$ and receiving all the inputs from tasks on the same group.

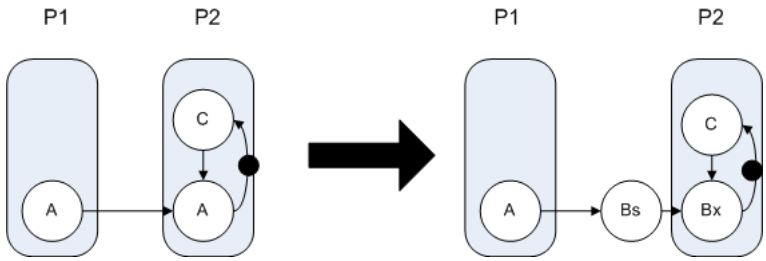

Figure 5.6: Actor separation to attain tighter throughput analysis.

It is now clear the meaning of the group tag on the dataflow graph description file. It allow us to define how to perform this transformation and how the groups are established. Groups must be formed only by tasks running on the same processor as the other members. Checks for that are enforced. The cost to perform this transformation is of polynomial complexity.

5.5 Maximum Cycle Mean Algorithms

We've seen how to obtain the MCM and throughput of a dataflow graph and how it's based on cycle detection. However, nothing was said on how that action is performed on the

Heracles tool. Some of the algorithms used by Heracles to calculate the MCM will now be presented. As the implementation of both algorithms was not part of the thesis (although changes were made to cope with the new CSDF data type), we will discuss them briefly, being, as they are, an important part of Heracles.

5.5.1 Howard and Szymanski

Both Howard and Szymanski algorithms are used to find the Maximum Cycle Mean of an HSDF graph. From the calculated MCM we can extract the maximum attainable throughput for the application. In order for an SDF or CSDF graph to be analyzed this way we must first find its HSDF equivalent since both algorithms operate on this type of graph. In practive, despite the required transformation to HSDF, the runtimes are usually quite low, in the order of the seconds or fractions of second to most real world application graphs.

On table 5.5.1 we may see the complexity of the used algorithms. These values and a comparison with some more MCM algorithms can be found on [11] where a detailed analysis is also presented.

Algorithm	Year	Time Complexity	Final Result
Howard	1960	$O(n^4 mWT^2)$	Exact
Szymanski	1992	$O(nm \ln(nWT))$	Aproximate

Table 5.1: Comparison of MCM algorithms

These results are for a directed graph $G = (V, E)$ with $n = |V|$ nodes, $m = |E|$ arcs. The algorithms require also an arc cost function ω and an arc transit time function τ from which we obtain the maximum arc cost W and the maximum arc transit time T.

Chapter 6

Symbolic Simulator

On the previous chapters we've seen a way to compute the SDF/CSDF throughput by converting it first into the equivalent HSDF graph. However, the conversion may lead to an exponential blowup on the number of actors (especially on certain graph topologies), which carries non-ignorable consequences to the throughput analysis algorithms. Case in point, it can make analysis by Howard's and Szymansky's algorithms excrutiatingly slow due to the long time spent on the HSDF conversion algorithm and on the analysis itself.

As an attempt to avoid the conversion penalty some state space exploration methods have been divised, such as those presented in [12]. These methods operate directly on the SDF by executing it's self-timed behavior. The *SDF3* tool presented on [15] outputs C++ code that when compiled and executed depicts accuratly the self-timed execution of the original graph. The presented results indicate that simulating the self timed execution whithout performing the potentially costly conversion to HSDF is fast and viable analysis technique. This led to our proposal of exploring the space-state applying the same methods. However, in order to better integrate the simulation with the already existing code base an architecture based on a discrete event simulator was instead proposed. This simulation allows the extraction of some important statistics, such as per channel maximum buffer usage.

6.1 State Space Analysis

The state of a dataflow graph is a pair (γ, v), composed by the state of each actor and the state of its FIFO buffers.

As we may have several actors running concurrently, an actor state v_a is the sorted multiset (one element per executing actor) of the remaining execution time for each of the actor firings. The buffer state γ_a is defined by the ammount of tokens stored within the channel.

Let us define a dataflow graph transition as the passage of a state into another, different, state. We may denote a transition as $(\gamma_1, v_1) \rightarrow^{\beta} (\gamma_2, v_2)$, where β denotes the type of the transition $\beta \in \{start, finish, update\}$. A start transition may only occur, according to the dataflow graph firing rules, when enough tokens are present on all of an actor's input FIFOs. A finish transaction is scheduled as soon as a start transition is, as we have knowledge of the

actor execution/response time. An update transition is issued when a pre-determinated actor executes so that we may update and compare the states.

A graph execution is therefore the set of transitions from $t = 0$ until $t = \infty$, where t stands for the simulation clock time. In [12] is shown (and we may verify it also on figure 6.2) that a consistent graph execution flow is composed of two parts. A transient phase consisting of a finite number of transitions, and a periodic phase, a sequence of transitions that is repeated *ad infinitum* (see fig 6.2).

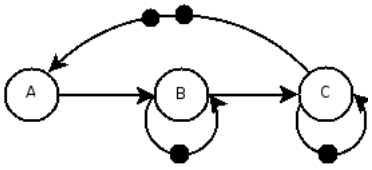

Figure 6.1: A simple dataflow Graph

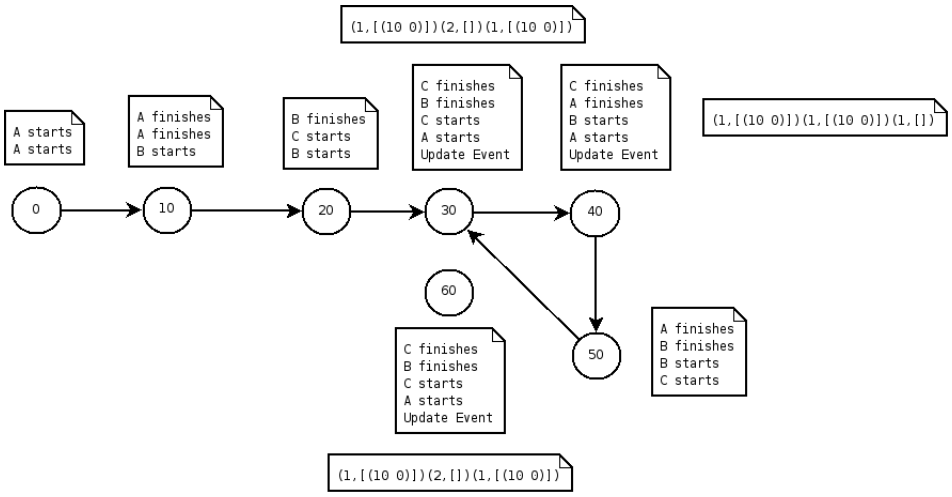

Figure 6.2: State Space exploration of the dataflow graph 6.1

When a new state, identical to a previous one, is detected we are in presence of the periodic phase of the graph's self-timed execution. It can be seen from the example on figure 6.2 that the state on $t = 60$ is identical to the one on $t = 30$. From there on the self-timed execution would repeat that cycle indefinitly.

6.2 Throughput Revisited

Throughput, as previously seen, is related to the MCM by being its multiplicative inverse. It can also be defined as the average number of firings of an actor a on the execution σ of a dataflow graph. Since we deal mostly with DSP algorithms that are supposed to run indefinetly we may say:

$$T_h(a,\sigma) = \lim_{t \to \text{inf}} \frac{fires(a,t)}{t}; \tag{6.1}$$

where $fires(a,t)$ represents the number of times actor a has been enabled, up until time t.

It follows that, if we focus on the periodic phase of the execution, repeating itself infinite times, the throughput of an actor a equals the average number of firings per time unit on that same phase. A more formal proof can be found on [12]. Having the throughput of an actor and the basic repetition vector of a graph $G = (V, E)$, we extract the throughput of the dataflow graph:

$$Th(G) = \frac{Th(a,\sigma)}{q_a} \tag{6.2}$$

The symbolic simulator executes the dataflow graph in a self-timed manner and the reported throughput is that of the graph's self-timed execution. It is demonstrated that the throughput obtained by state space exploration of an SDF graph G is identical to the throughput obtained by computing the equivalent HSDF graph H [12].

6.3 Implementation

As previously referred, the implementation of the graph execution was based on a discrete event simulator architecture. That is to say, a simulation clock (clk) is maintained and updated whenever a transition happens, events are issued, and the current state of execution is kept tracked.

We have 3 types of transitions, or events:

- Start - Issued whenever an actor has conditions to fire.
- Finish - Issued at the same time of Start, but with a different event time (clk + execution time)
- Update - Issued when we wish to commit the current state to memory, updates the state table and performs the cycle detection.

Our discrete event simulator is composed of a clock, an ordered event queue, and hashtable to keep the states and perform the cycle detection, needed to discover when we've entered the periodic part of the execution. An event has the following fields:

- Event Time - When is this event activated.
- Priority - So that we have finishes before starts in a consistent manner
- Issue Number - So that we may sort events with the same priority in a FIFO policy
- Issuer - Actor that issues the event

The event queue is sorted firstly by Event Time, events with the same Event Time, are sorted amongst themselves by Priority. Thus, Finish Events, having higher priority than Start Events, will be computed before the latter. This is in order to update the buffers and check if more Start events can be issued. Should events have the same Event Time and Priority, they are sorted by issue number. The sooner an event is issued, the lower its issue number. This allows for a FIFO policy for similar events.

In order to calculate the graph's throughput we need the repetitions vector, the duration of the periodic phase of the self-timed execution and the number of firings of an actor during that same phase. All of which are easily obtained. The fact that we only need a single actor throughput suggests an important optimization. There is no need to keep track of every single state. As such, we choose a single actor to issue Update events and only then the state is kept and checked for recurrency. This allows for a reduced memory footprint and greater speed. Furthermore, Heracles selects to issue Update events the actor with the lowest value on its repetitions vector.

At the beginning of the simulation, ($clk = 0$), we search for ready to fire actors (the ones already having tokens on their input edges due to delays) and issue Start events for those actors with $EventTime = 0$. That is all it takes to get the simulator running. As the Start events are pulled from the event queue Finish events are then added to that queue with $EventTime = clk + executiontime$. The simulation clock jumps as events are removed from the event queue. When a Finish event is detected, and the token distribution updated, we check whether any actor my start. We restrict that search on the actors connected to the actor that finishes, as no other actor may start. If an actor starts, Start events are added to the queue and the cycle repeats itself. Should the simulator, at any time, be left without any event in the queue, then a deadlock has been found. Consult figure 6.2 for an execution example.

In order to detect whether a state is recurrent an hash table is used. The state is periodically added to the hash table that is currently limited to 10000 positions. The behavior of Hash tables in ocaml allows for multiple states to share the same hash position and no effort has been made to guarantee that no collisions will happen between states. As such, when a collision is detected it must be checked whether the state is in fact a duplicate of some previous state. If indeed a duplicate state is found the periodic phase of the execution has been found.

6.4 Buffer Size Calculation

During the graph's execution we must control the flow of data (tokens) in order to know when an actor should fire. As such, we may as well check other buffer statistics, as doing it is practically free since we already have all it takes implemented. So we keep track of a channel maximum utilization, under conservative conditions. In order to be conservative in the buffer size simulation tokens are placed in the exit buffers as soon as an actor starts executing (at the Start event) and removed from the input buffers only when an actor finishes executing (Finish event). Thus tokens remain in buffers probably a bit longer than what would happen in a real execution, but this gives us upper bounds we can trust.

Chapter 7

Scheduling and Mapping

This chapter describes the parallel scheduling of applications onto multiprocessor systems and respective implementation onto the Heracles tool.
Within the context of multiprocessor applications, scheduling is defined as the order of execution that the various tasks follow in order to complete successfully. Note that the application might, as it is the case with most DSP applications, execute the schedule an indefinite number of times.
In general, to properly define the scheduling problem let us assume three sets, a set $J = \{j_1, j_2, ..., j_n\}$ of n tasks, a set $P = \{p_1, p_2, ..., p_m\}$ of m processors and a set $R = \{r_1, r_2, ..., r_k\}$ of k resources. Scheduling is the process of assigning processors from P and resources from R to tasks from J in order to complete all tasks under the imposed constraints.[17]

The set of tasks and their precedences is specified by a given dataflow graph and the processor set by an MPSoC system file. There is currently no way to specify buffer or latency constraints that a scheduled application must respect, although both the maximum buffer usage and latency constraints can be hard coded into the dataflow graph. In order to express buffer constraints we can apply back-pressure[9] between the actor consuming tokens and the actor producing them. We can also simulate the scheduled execution of the application afterwards with the symbolic simulator. Latency constraints can be expressed as throughput constraints as shown on [9].

The metric used for performance evaluation of a scheduled application is its throughput μ_g, or the equivalent MCM. Thus, an optimal schedule is one that maximizes μ_g (or alternatively, one that minimizes the MCM).[4]

7.1 Scheduling Strategies

There are several possible types of scheduling strategies available to use with a given application, each one with its tradeoff between run-time computation costs and the generality of its application.
In general, the more we know about an application behavior the more decisions can be made during compile time. Reciprocally, the more run-time decisions we make, the less we need to know about the application specifics and the more general that strategy is. Alas, such a dynamic strategy can use a lot of resources needed for the application itself.

Since we're dealing with hard real-time applications under somewhat limited systems, it is advantageous to reduce to a practical minimum the run-time computation costs thereby performing as much of the scheduling as possible during compile time.

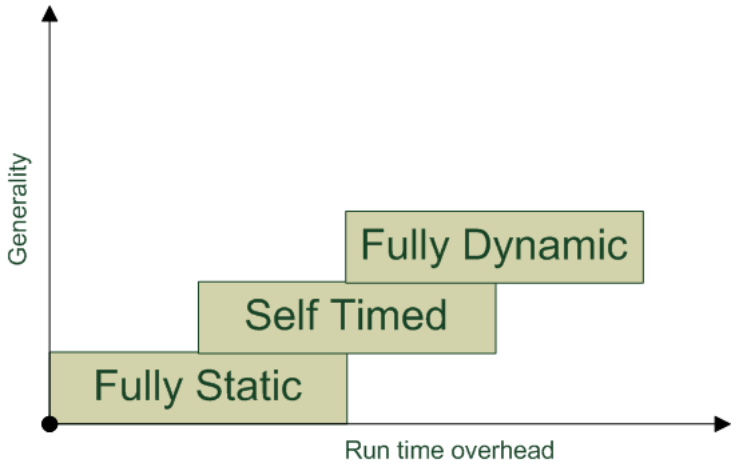

Figure 7.1: Various scheduling policies.

7.1.1 Fully Static Schedules

In a fully static strategy a very precise execution time for each task must be known beforehand in order to exactly specify each task's firing time and guarantee sender-receiver synchronization. During each clock cycle the processor's state is fully specified, and both starting and finishing times of tasks are hard-coded beforehand into the processor scheduler. Should a task finish before another one is scheduled to start, nop instructions are issued[17]. The problem of creating an optimal fully static schedule is NP-Hard[19].

7.1.2 Self Timed Scheduling

To cope with variable or not precisely defined execution times a solution is to introduce explicit synchronization whenever tasks communicate. This leads to a self timed scheduling strategy where only the processor assignment and order of execution of each task is determined. Precise timing information (start times and finish times) is no longer required.
Exactly when a processor executes an actor will depend on when, during run-time, data for that actor is available. This is unlike the fully static case, where hardly any run-time check is needed. Conceptually, an executing actor writes data to a FIFO buffer and blocks whenever that buffer is full. On the other hand, the receiver will block if there is no data on the buffer[4].

7.1.3 Dynamic Scheduling

In a fully dynamic strategy, all scheduling decisions are perform during run-time. This approach will handle highly variable program behavior by dynamically changing the ordering on which the tasks are run and adjusting, if required, the processor loads.
However, since decisions are made during run-time, its simply unpractical for the scheduler

to make globally optimal decisions (there is a high degree of computational complexity involved) and must resort to greedy locally optimal decisions. So, in presence of compile-time information, such as execution times and precedence constraints, both static and self-timed strategies will almost certainly yield better performance. This is, nonetheless, a very general approach that works on a broad range of applications and processors.

7.2 Multiprocessor Scheduling Complexity

It is desirable to find an optimal schedule with respect to some criteria be it to minimize resource usage, maximize throughput or something else. How hard is it then to find such a schedule? The problem of finding an optimal schedule according to a specific criteria can be formulated as a bin packing problem. This means the we are facing an NP-Hard problem[19] and, as such, for a certain size and topology of the application graph, it becomes prohibitive to search the optimal solution through all of the possible solutions. Some algorithms of polynomial complexity have been proposed[16] and [13] based on First-fit Vector Bin Packing, although of course, there are no warranties that the solution offered by those algorithms is an optimal one.

7.3 Scheduler Implementation

Scheduling an application graph to exploit task parallelism involves a few steps. One such step is to attribute a processor for the task at hand, this is the so called mapping process or processor assignment step. The other step is to assign an order of execution to the tasks running on a processor such that all the data precedence constraints are met and no deadlocks are created. This is the actor ordering step.

The scheduling process generally will introduce some precedence dependencies, not present on the original application graph, due to the restrictions that the limited amount of resources may impose on the natural parallelism of the graph. Furthermore, in general, different schedules will yield different throughputs and dissimilar buffer capacity requirements. Given that embedded systems usually have very limited resources available to run the various tasks at hand, it is understandable the desire to find a schedule that will minimize both the MCM as well as memory consumption. An ideal schedule will be the one that will not force additional precedence constraints to the application graph and will therefore have the same throughput as that of the self-timed execution of the application graph. Of course, that could require a large amount of resources.

Let us use the application graph from 7.2 as an example to show how scheduling may affect the application throughput. Each task has an execution time of 1 time unit, there are two possible cycles (ABDA and ACDA) both yield an MCM that equals 3 time units. The tasks C and B can run in parallel. However, if we have only one processor to map this application unto (or they must be mapped on the same processor), that condition no longer holds, and C and B cannot run at the same time. Either B or C will have to run first. This lost parallelism will be represented on the graph by an extra precedence dependency from the first running task to the second. In 7.3 we choose to let B run first and a dependency is added from B to C. In this case, the order in which the tasks would run would be: ABCD. The added dependency

has consequences for the MCM. The largest cycle is now ABCDA and the MCM is now of 4 time units.

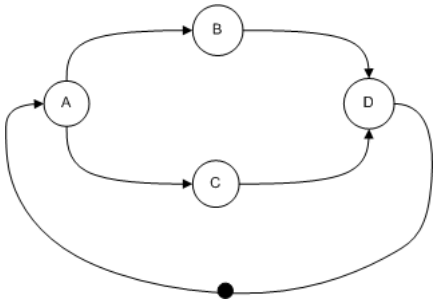

Figure 7.2: A simple HSDF application graph.

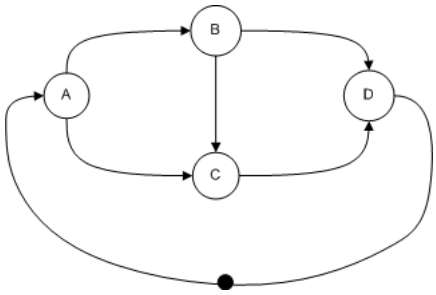

Figure 7.3: The graph from 7.2 with a dependency added from scheduling. Notice that the execution order must now be A → B → C → D, due to the fact that all task dependencies must be fulfilled before a task might be enabled.

Let tasks A,B and C execute on processor P1 and D execute on processor P2. The following scheduling can be applied:

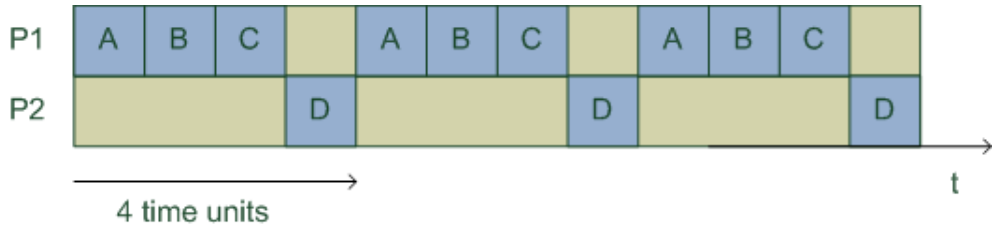

Figure 7.4: A repeated schedule.

The schedule on figure 7.4 is a blocked schedule since successive iterations of the HSDF graph are treated separately. Each iteration must finish before the next one begins. The HSDF is scheduled as if executing only for one iteration of the graph, and then that schedule is repeated to obtain an infinite periodic schedule. The length of the block determines the throughput of the application (also called the makespan of the schedule). By applying precedence constraints (as edges on the dataflow graph) in order to model the scheduling

precedences worst-case response time analysis can be applied to obtain the application's MCM and therefore it's expected throughput.

The scheduling algorithm of Heracles constructs a makespan self-timed schedule for a given dataflow graph. It is implemented as a list-scheduler with full backtracking capabilities. The actor list is created using a DAG that is extracted and updated on each step of the scheduling process. The initial DAG is easily created by removing edges that represent dependencies between successive iterations (i.e. edges with delays). The DAG's actors without precedencies are the ones that are ready to fire. Let's analyze a simple example assuming the existence of 2 processors P1 and P2 that can execute any of the graph's tasks. We wish to obtain a schedule with a production of at most 2 time units.

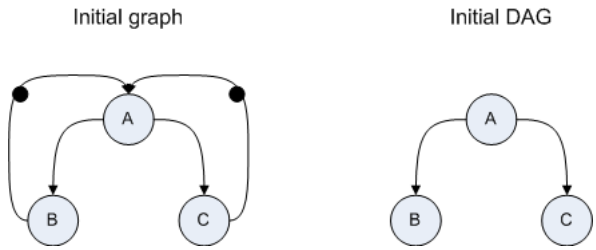

Figure 7.5: Step 1 of the scheduling algorithm. On the right the graph's respective DAG.

On the first step a DAG is extracted from the original graph. The only actor that is ready to fire is A, due to the lack of precedence edges. The scheduler returns an actor list with A as its single element. We now proceed to find the list of processors where A can be mapped (processors whose type must be the same as A's type, given in the system and graph input files). That list is composed by both P1 and P2. The scheduler now associates the first element of the actor list to the first element of the processor list. We check if the constraints are met and proceed to the next step.

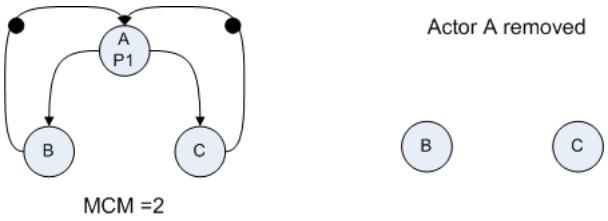

Figure 7.6: Step 2 of the scheduling algorithm.

Steps 2 and 3, like the first, consist of removing the previously scheduled actor from the DAG, creating a list of ready to fire actors. For the first actor on the actor list, a list of valid processors is created, the first actor is mapped on the first available processor and the dependencies are added to the scheduled graph.

However, on step 4 the graph exceeds its maximum allowed MCM. We must now backtrack. In the example we return to the solution of step 3, but no longer do we consider the list of valid processors to be composed of P1 and P2. We've already seen that mapping actor C to

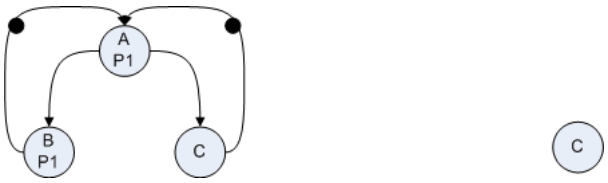

Figure 7.7: Step 3 of the scheduling algorithm.

P1 fails. So we map C to the next processor of the returned list, that is P2. That solution respects the imposed constraints. If it didn't, the scheduler would backtrack to step 2 and would try to map C before B.

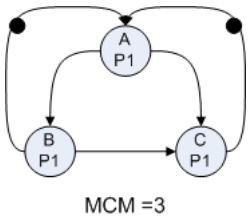

Figure 7.8: Step 4 of the scheduling algorithm.

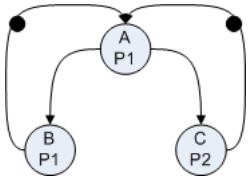

Figure 7.9: Final step of the scheduling algorithm.

We have seen that several possible schedules may exist for the same application graph and MPSoC. Different schedules present different tradeoffs between throughput and memory consumption. As such, it is desirable that we may search for a good enough solution through the various possible solutions. However, the very nature of the problem makes such a search infeasible for all but the most trivial graphs. To deal with this problem a mixed strategy was followed. The scheduler is capable of presenting multiple solutions in order to let one analyze the different tradeoffs and choose the most suitable schedule. Also, it is possible to reduce the scope of the search by binding a task to a processor beforehand (using the map tag on the CSDF file descriptor), or grouping tasks that must be on the same processor before the search is initiated (using the group or comap tag). Furthermore, the scheduler performs branch pruning. If we provide the scheduler with a maximum MCM that must not be exceeded, on each step of the scheduling we match the currently obtained MCM against the maximum possible. If it exceeds that value we backtrack to the previous acceptable solution, discarding all of the solutions that derive from the discarded one, and try other mappings and orderings from there. This means we do not schedule the remaining tasks, something that would be pointless since we already know that the partial scheduling is invalid, allowing us to decrease the search space.

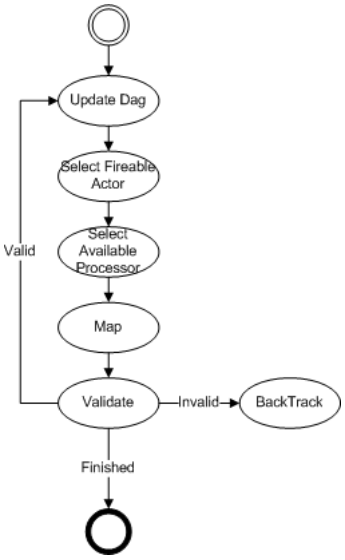

Figure 7.10: A dataflow diagram for the scheduling algorithm.

7.3.1 Connection Model and Map tag

By allowing us to give hints to the scheduler, such as telling it that task B and C must run on the same processor as task A, we can further decrease the search tree. This hint can be given with a comap tag on the dataflow graph file descriptor. The group tag will also have the same effect. From the moment we schedule task A, we already know that tasks B and C will only run after task A. We can then add a dependency from A to B and from A to C. In some cases this will increase the MCM of the scheduled graph. If it happens to exceed the maximum MCM we can prune that search branch more quickly and proceed to search other valid solutions.

To further prune the search tree there is the map tag in the dataflow graph file descriptor. Often a task is bounded to a specific processor since inception. A signal analysis task will probably ran only on a vectorial processor. If there is such information we can greatly decrease the search space and therefore speed up the search for valid solutions. This can be achieved with the map field on the file descriptor file.

Chapter 8

Deadline Extensions

8.1 Deadline Extension Pool

For purposes of adjusting a task slice time in order to release resources not strictly needed by the application, or to have a general idea of the application behavior, it is useful to know how much slack a task not in the critical cycle has. Also, should the need arise, during prototyping, to add more tasks to the dataflow graph it can be convenient to know where to add them while causing minimum disturbance in the graph throughput (it is, of course, not always possible to add a task where we'd like. The task to add might have dependencies on previous tasks that must be fulfilled). This can also be achieved by looking at the available slack. More formally, the amount of slack is known as a task deadline extension, and is defined as the amount by which a task's response time can be increased, while response times for other tasks are kept constant and still guarantee that the MCM of the application is kept below or equal to a maximum desired production period.[9] Tasks in the critical cycle have, by the very definition of it, no slack, unless we specify a lower throughput (or conversely, an higher MCM) for the application other than the one obtained by analysis of the dataflow graph. Should any task have a negative deadline extension we can conclude that it does not respect our MCM constraints and as such we have another method to check whether an application respects its constrictions.

On figure 8.1 is presented a dataflow graph with an MCM of 25. Its critical cycle is composed by the nodes $C = \{A, B, E, G\}$. The deadlines are showed on the nodes. Given this graph we could increase the response times of the tasks $D = \{C, D, F\}$ without affecting the application performance. One way we could achieve this would be by decreasing the task slice time, allowing more resources to other applications, therefore putting our multiprocessor system resources to better use. It is also useful data for the designer to know in order to know where to apply optimizations, if required.

By performing some transformations on the problem set, as depicted in [1], we can transform the problem of finding the maximum deadline extension of all actors into an instance of the Floyd-Warshall all pairs shortest path algorithm with a cost function $w(i, j) = \mu_D \cdot d(i, j) - r_t(i)$ applied, where μ_D stands for the desired MCM, $d(i, j)$ for the number of delays in edge (i, j) and $r_t(i)$ accounts for actor's i execution time. This algorithm has a polynomial complexity of $O(n^3)$ [20].

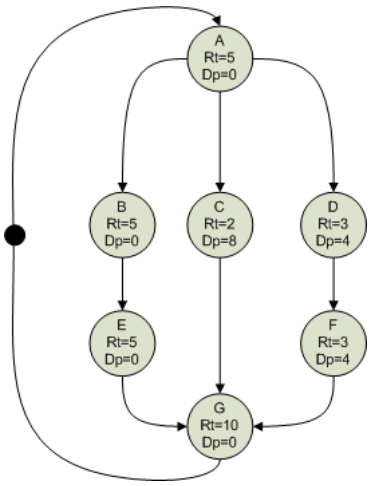

Figure 8.1: A dataflow graph with its respective deadlines

8.2 Deadline Optimization

One way to distribute the slack amongst actors is to express it as a sum of deadline extensions that can be maximized regarding a predefined linear cost function. This is an instance of the inverse shortest path problem[1], where weights have to be chosen in such a way that no actor has a negative shortest path to itself. That implies the presence of a cycle whose throughput constraints have been violated. The formulation of the problem can be presented as a linear program where for every actor $v \in V$ and every edge $(u, y) \in E$ it must hold, due to the definition of shortest path, that $\sigma(u, y) \leq \sigma(v, u) + w(u, y)$. If the weights are $w(u, y) = \mu_D \cdot d(u, y) - t(u) - \delta(u)$, the linear program is:

Maximize	$\sum_{\forall v \in V} \delta(v)$
subject to	
$\forall (u, y) \in E, \forall v \in V,$	$\sigma(v, y) \leq \sigma(v, u) + \mu_D \cdot d(u, y)$ $-t(u) - \delta(u)$
$\forall v \in V,$	$\sigma(v, v) \geq 0$
$\forall v \in V,$	$\delta(v) \geq 0$

Figure 8.2: A generic linear program to optimize sums of deadlines.

8.3 Finding Slice Times Through Deadline Optimization

The linear program presented on the previous section cannot be directly used to figure out possible slice times for grouped tasks (as described on section 5.4) running on TDMs. The problem is that our response time equation 5.4 is non linear due to the ceiling function and therefore cannot be directly inserted into the provided system of equations to obtain the response time. We can however linearize 5.4 by making some conservative approximations (read overestimative approximations). We arrive then at the following linear equations for a

task's response time:

$$r_{tl}(v) \leq t(v)/S(v) \cdot P(\pi(v)) \tag{8.1}$$

where $t(v)$ is an actor execution time, $S(v)$ is the allocated slice time to actor v and $P(\pi(v))$ is the time wheel period of the processor where the actor has been mapped. Recall that if an actor has a token production across different processors, the response time equation contains two elements becoming then:

$$r_{tl}(v) \leq t(v)/S(v) \cdot P(\pi(v)) - P(\pi(v)) \tag{8.2}$$

A processor usage $U(p)$ is calculated dividing the total slice time allocated to it by its time wheel period. We then proceed to perform a variable substitution: $N(p) = U(p)^{-1}$ and add an extra constraint: $N(p) \geq \hat{\mathrm{U}}$. This value allow us to specify what the maximum usage for a processor should be in the obtained solution. We choose to maximize a weighted sum of $N(i)$. The weights c allow us to better control the processor load, since some processors might be more required than others and their utilization should be kept lower. Processor weights are specified by the height field on the system description file.
The processor assignment of actor v to a processor from the set of available processors Π is represented as $\pi(v)$. The dataflow graph may or may not be scheduled, but each process must already have a processor assigned otherwise response times would be undetermined.
For a mapped dataflow graph $G = (V, E)$ we then arrive at the following system of equations[1]:

Maximize	$\sum_{\forall v \in V} N(\pi(v)) \cdot c^{-1}(\pi(v))$
subject to	
$\forall (u,y) \in E : \pi(u) = \pi(y)$,	
$\forall v \in V$,	$\sigma(v,y) \leq \sigma(v,u) + \mu_D \cdot d(u,y)$ $-N(\pi(u)) \cdot t(u)$
$\forall (u,y) \in E : \pi(u) \neq \pi(y)$,	
$\forall v \in V$,	$\sigma(v,y) \leq \sigma(v,u) + \mu_D \cdot d(u,y)$ $-N(\pi(u)) \cdot t(u) + P(\pi(y))$
$\forall v \in V$,	$\sigma(v,v) \geq 0$
$\forall p \in \Pi$,	$N(p) \geq \hat{U}(p)^{-1}$

Figure 8.3: System of linear equations for the Linear slicer.

Implementation

For the implementation of this algorithms the Glpk library, as well as its Ocaml bindings, were used.

Chapter 9

Pos-optimization of slice times

After an application is scheduled and mapped we may wish increase the response time of tasks not in the critical cycle. We want to do this in order to give more processor time to some other application running on the same processor. Furthermore, we may even increase response time of tasks in the critical cycle if we verify that the current throughput is higher than what is strictly required. This chapter shows some of the algorithms implemented in Heracles to deal with that situation.

9.1 Binary Slice Allocator

This is a very simple algorithm, presented in [14] and is implemented firstly on the *smart* tool and now on Heracles. If a given scheduled dataflow graph respects a specified throughput constraint when using some pre-allocated time slices (usually the entire processor time wheel), this algorithm will find new slice values for each group that will be smaller or equal to the pre-allocated ones will still respecting those same throughput constraints. This will increase some tasks response time, effectively lowering the throughput of the application closer to the minimum specified.

The gist of this algorithm is a binary search that is performed on each of the scheduled groups of tasks. Groups are sorted according to the priority of the processor where they are mapped. Afterwards, starting with the group with the highest priority, we calculate the new slice time using a binary search between the possible slice values. A pseudo-code representation is as follows:

```
binary_search (min_slice, max_slice, last_valid_slice)

        if check_stop_condition /*max_slice>min_slice*/
                return last_valid_slice

        new_slice = (max_slice-min_slice)/2
        new_mcm = compute_graph_MCM (new_slice)

        if (new_mcm > max_mcm)
                bin (graph, new_slice+1, max_slice, las_valid_slice)
        else
```

```
                    bin (graph, min_slice, new_slice-1, new_slice)
end
```

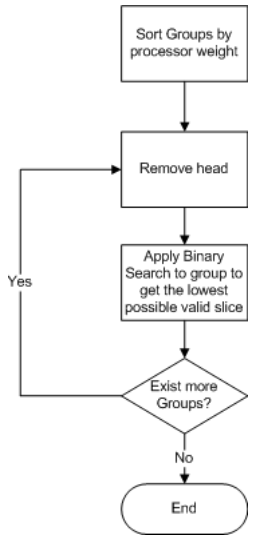

Figure 9.1: Flowchart for the binary slice allocator.

The results for this algorithm, however, will depend heavily on which processor and group we start the binary search. In practical terms, what usually happens, is that the algorithm tries to lower to the maximum the slice time for the first groups it performs the binary search on, leaving no slack that allows the remaining groups to use to lower their slice times. That is the reason groups are prioritized, in order to allow the designer to select on which groups the search is first applied.

This algorithm being a combination of a binary search and the cycle detectors, both algorithms of polynomial complexity is also itself on the P class of algorithms.

9.2 Random Slice Allocator

It was observed that the binary slice allocator depends quite heavily on the group ordering on which the search is performed. To overcome that weakness this algorithm randomly selects groups to decrease their slice time by a specific amount, thereby trying to be fair to all groups. As in the previous algorithm, if the scheduled graph respects the specified throughput for some predefined slice times, this algorithm will find smaller or equal slice times for the graph thereby decreasing the throughput to as close as possible to the specified minimum throughput.

The random slice allocator adds all groups to an open list (i.e. a list of groups where we may still decrease the slice) and will select a group from there. That group will see its slice time decreased by a specified amount and the graph is tested for compliance with the throughput constraints. Should the smaller slice result on a violation of imposed constraints, that group

is removed from the open list and its slice value restored to the previous amount. This select and decrease step is repeated until there are no groups on the open list, which means it is not possible anymore to lower slice times while respecting the throughput constraints.

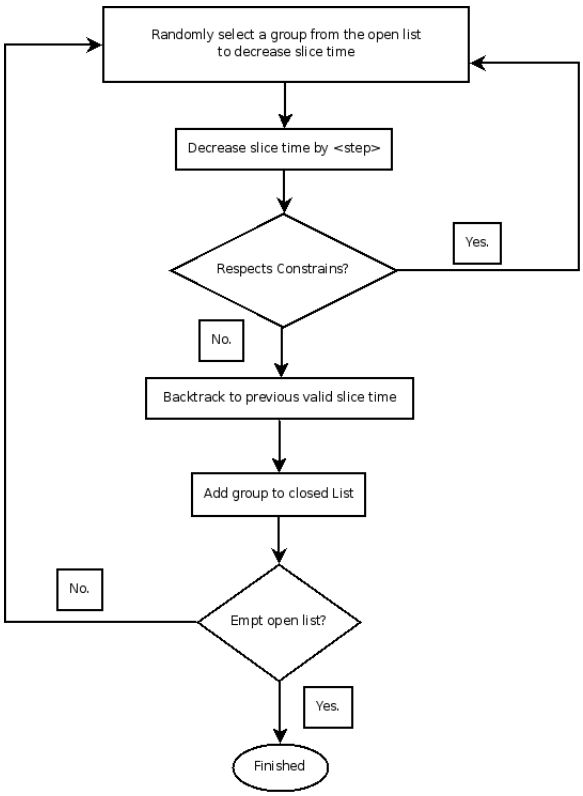

Figure 9.2: Flowchart for the random slice allocator.

This algorithm depends also on a cycle detector. Essentially it runs the cycle detector a fixed number of times depending on the step size and on initial group's slice times. The maximum running time for this algorithm can be bounded and is also a class P algorithm.

Chapter 10

Case Study

In this chapter some of Heracles key functionality is demonstrated by means of a simple walkthrough. The presented WLan and TD-SCDMA applications are to be analyzed and scheduled by Heracles onto a MPSoC architecture composed by a special purpose Software Codec processor combined with an EVP and an ARM processors.

In this example we wish to obtain scheduler settings that allow for the concurrent execution of both applications on the same MPSoC. Figures 10.1 and 10.2 show the dataflow graphs of those applications. The executions times of the various tasks, indicated under the actor nodes, are given in nanoseconds. Nodes whose names start by "Src" model the input obtained by an external RF unit. Latency nodes (LatencyHeader and LatencyPayload) were added to the model in order to convert latency constraints into throughput constraints.

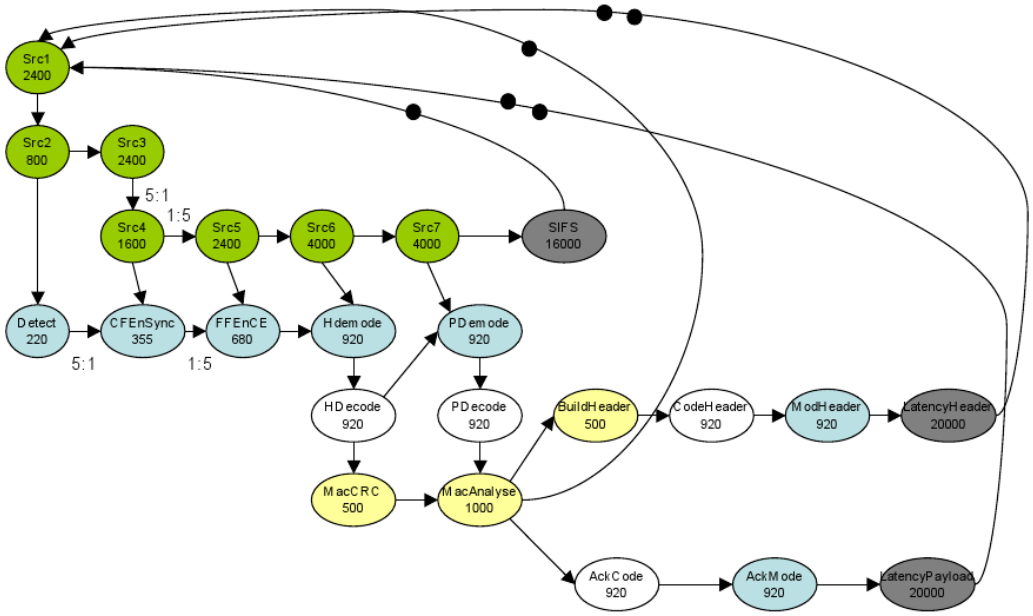

Figure 10.1: A Wireless Lan 802.11a decoder

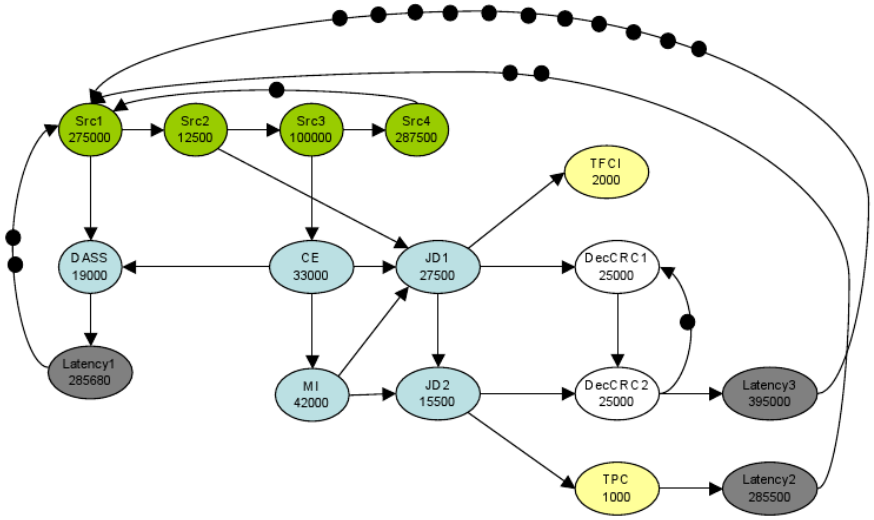

Figure 10.2: A TD-SCDMA job

The first step is to create the files to be parsed by Heracles with the description of both applications and the MPSoC. Those can be consulted on the Appendix section.

We can then determine the self-timed production period, given by the graph MCM. This value is of $\mu_{wlan} = 40000$ for the WLan and of $\mu_{td-scdma} = 675000$ for the TD-SCDMA. From the MCM of both graphs we conclude that the maximum throughput possible is $T_{wlan} = 2.5 \times 10^{-5}$ and $T_{td-scdma} = 1.481481 \times 10^{-6}$

These values are obtained by executing Heracles as follows:

```
./Heracles -f wlan-pay1.sdf -c2h -how
./Heracles -f tdscdma.sdf -how
```

The -f "filename" switch specifies the path to the application dataflow file to be parsed. The -c2h switch makes Heracles perform a conversion from a CSDF into an HSDF. This is required since, as seen, the cycle detection algorithms operate only on HSDF graphs. WLan, being an SDF graph, needs to be converted to a more workable form. Finally, the last switch -how invokes howard, the cycle detection algorithm, and that will return the graph's MCM.

Since we wish to both applications run simultaneous, it is a given that the sum of slice times for the various tasks of an application running on the same processor mustn't exceed half of the processor's time wheel. However, we wish to spend less than 10 % of time context switching. Each application can therefore use only at most slices occupying 45% of the processor time. A worst case context switch time of 50ns for the EVP and Software Codec processors and of 100ns for the ARM is considered a safe assumption[1]. This yields time wheels of 1000ns for EVP and Software Codec processors and of 2000ns for the ARM.

It is now desirable to know, before a schedule is obtained, how will both applications behave in face of the changes introduced to deal with worst case response times. As mentioned, the

Src nodes model an external input, and therefore are impervious to response time changes since those actors are not to be scheduled. Likewise regarding Latency actors. Since we know which tasks were designed to run on the EVP and which tasks are meant to run on ARM and on Software Codec we apply the map tag on the dataflow file and proceed to calculate a response time model as described in chapter 5.4. That is accomplished by running Heracles as follows:

```
./Heracles -f tdscdma.sdf -s tdscdma.sys -rmcm
./Heracles -f wlan-pay1.sdf -s wlan.sys -rmcm
```

The -s "filename" switch specifies the path to a MPSoC system description file and the -rmcm switch instructs Heracles to calculate the response time model and calculate its respective maximum cycle mean. For this example, both system files are identical. The results show that the resulting MCM is equal to the previously obtained MCM of the self timed execution. This is due to the fact that the more expensive cycles are composed by Source actors. They are not affected by the response time transformation and, since they compose the largest cycles on both graphs, they determine the throughput.

Now we can proceed to schedule the applications. Since there is no need to schedule Source and latency actors and the remaining actors have been pre-mapped to specific processors that is a trivially quick task for the implemented scheduler to accomplish. Self-timed schedules can be obtained by issuing the following commands:

```
./Heracles -f tdscdma.sdf -s tdscdma.sys -scc -how
./Heracles -f wlan-pay1.sdf -s wlan.sys -scc -how -c2h
```

The -scc switch indicates Heracles to search for a schedule with a maximum production period as given by -how. This means a schedule is wanted whose throughput is identical to that of the self-timed execution of the dataflow graph. Another (larger) production period could be specified with the switch -th. However, since the bottleneck of both applications is, as seen, in the Source actors which are not to be scheduled a schedule is obtained with the same production period of the self-timed execution of the dataflow graphs.

After the schedule is obtained the slice times attributed to the tasks running on the various processors can be optimized. This is done applying the algorithms described on section 8.3 and on chapter 9.
Applying the linear deadline optimization algorithm having has goal a processor utilization of at most 45%, $N(p) \geq 1/0.45$ we obtain the results from tables 10.1 and 10.2.

Weight			Utilization		
EVP	ARM	SwC	EVP	ARM	SwC
1	1	1	0.449	0.449	0.209
2	1	1	0.276	0.449	0.449
2	2	1	0.276	0.449	0.449
1	2	1	0.449	0.276	0.449

Table 10.1: Processor use with scheduled WLan application.

Weight			Utilization		
EVP	ARM	SwC	EVP	ARM	SwC
1	1	1	0.449	0.15	0.13
100	1	1	0.429	0.15	0.13
1000	1	1	0.328	0.15	0.449

Table 10.2: Processor use with scheduled TD-SCDMA application.

The weight columns represent the cost value associated with the processor in the function to maximize. Changing costs allows for different tradeoffs yielding different slice values and therefore different utilization of the processors.

Applying the pos-optimization of slice time algorithms to the scheduled graphs we obtain the following results:

Weight			Group Slices			Utilization		
EVP	SwC	ARM	EVP	SwC	ARM	EVP	SwC	ARM
3	2	1	326	15	667	0.326	0.015	0.3335
2	1	1	326	15	667	0.326	0.015	0.3335
1	2	1	443	15	22	0.443	0.015	0.011
1	1	2	443	15	22	0.443	0.015	0.011

Table 10.3: Processor use and slice times on scheduled TD-SCDMA application after pos-optimization with binary slice allocator.

Step	Group Slices			Utilization		
Step	EVP	SwC	ARM	EVP	SwC	ARM
1	230	230	500	0.23	SwC	0.25
15	210	330	660	0.21	SwC	0.33
20	190	370	780	0.19	SwC	0.39
25	325	200	500	0.325	SwC	0.25

Table 10.4: Processor use and slice times on scheduled WLan application after pos-optimization with random slice allocator.

In order to apply the referred optimizations the -nbslc switch must be used if the binary slice allocator is to be used and -nslc2 for the random slice allocator.

```
./Heracles -f tdscdma.sdf -s tdscdma.sys -how -scc -nbslc
./Heracles -f wlan-pay1.sdf -s wlan.sys -c2h -how -scc -nslc2
```

Lets now look at the buffer sizes. In order to do so the symbolic simulator of Heracles is executed on the scheduled dataflow graphs.

```
./Heracles -f tdscdma.sdf -s tdscdma.sys -how -scc -sim
./Heracles -f tdscdma.sdf -s tdscdma.sys -c2h -how -scc -sim
```

This action yields the results presented on figure 10.3.

```
Source: R320     Destiny: R321   1 0 1
Source: R321     Destiny: R322   1 0 1
Source: R322     Destiny: R323   1 4 5
Source: R323     Destiny: R324   1 2 5
Source: Detect   Destiny: CFEnSync       1 4 5
Source: CFEnSync         Destiny: FFEnChEst      1 1 5
Source: CFEnSync         Destiny: CFEnSync       1 1 2
Source: R324     Destiny: RHeader        1 0 1
Source: RHeader  Destiny: RPayload       1 0 1
Source: RPayload         Destiny: RPayload       1 1 2
Source: RPayload         Destiny: SIFS   1 0 1
Source: SIFS     Destiny: R320   1 0 1
Source: R321     Destiny: Detect         16 0 1
Source: R323     Destiny: CFEnSync       16 1 2
Source: R323     Destiny: R323   1 2 2
Source: R324     Destiny: FFEnChEst      16 0 1
Source: FFEnChEst        Destiny: HDemode        1 0 1
Source: RHeader  Destiny: HDemode        40 0 1
Source: HDemode  Destiny: HDecode        13 0 1
Source: HDecode  Destiny: PDemode        1 0 1
Source: RPayload         Destiny: PDemode        40 0 1
Source: PDemode  Destiny: PDecode        45 0 1
Source: PDecode  Destiny: MacAnalyse     9 0 1
Source: HDecode  Destiny: MacCRC         2 0 1
Source: MacCRC   Destiny: MacAnalyse     1 0 1
Source: MacAnalyse       Destiny: SendPreamble   2 0 1
Source: MacAnalyse       Destiny: BuildHeader    2 0 1
Source: MacAnalyse       Destiny: AckCode        2 0 1
Source: BuildHeader      Destiny: CodeHeader     2 0 1
Source: CodeHeader       Destiny: ModHeader      4 0 1
Source: ModHeader        Destiny: SendHeader     80 0 1
Source: SendHeader       Destiny: LatencyHeader  0 1 1
Source: LatencyHeader    Destiny: R320   1 1 2
Source: AckCode  Destiny: AckMod         1 0 1
Source: AckMod   Destiny: SendPayload    80 0 1
Source: SendPayload      Destiny: LatencyPayload         0 0 1
Source: LatencyPayload   Destiny: R320   1 1 2
Source: SendPreamble     Destiny: R320   1 0 1

shared: 60
shared with token_size: 426
internal:       60
internal with token_size:       426
Steps:  323             Clock:  128040
Periodic cycle: 40000
fires:1         clock:128040    tpstart:88040
LatencyPayload updater total fires:0
total fires:0
Th:0.000025             MCM:40000.000000 26 1
```

Figure 10.3: Simulated WLan application.

The symbolic simulator presents the throughput and the MCM as well as some other statistics of the simulation. More interestingly it presents the buffer statistics. For each FIFO channel the the tool outputs the token size, followed by how many tokens the channel was holding at the end of the simulation and more importantly the maximum number of tokens that have been simultaneous on the buffer. As each token can have its size in bytes the total amount of buffer space used is also calculated. The values are of 426 units for the WLan application and of 20 units for the TD-SCDMA application.

Chapter 11

Conclusions and Further Directions

We've presented and implemented several static time analysis techniques (some of them new) ranging from scheduling to pos-scheduling slice time analysis. They provide an important help in the quest to give proper guarantees regarding the application behavior and assure that no deadlines or constrictions are violated. We can conclude that static time analysis is indeed a valuable tool (although not the only) in the large and moving fields that are multi-processor systems and concurrent applications. However, what has been presented is but a small subset of what is currently being research on the area of multi-processor systems.

Even Heracles by itself has the potential to be much more complete. For most algorithms (scheduler, pos-optimization of slice times) we have focused mostly on throughput constraints. These are not the only type of constraints. Many more exist that are not yet taken into account by the Heracles tool (at least automatically, some of them can be expressed directly on the analysis model), such as sizes for the FIFO buffers, latency and energy consumption. A goal for further investigation and implementation would be to allow for such constraints to be properly expressed. Another goal was to streamline the design of concurrent applications. Still sometimes, custom tweaking of the tool is necessary to achieve some goals. This could be improved. Also we've considered only the TDM model for intra processor scheduling. More types of intra-processor scheduling might be studied and implemented. Regarding scheduling, we've only considered makespan static and self timed schedules, more technics exist to allow for a lower throughput like unfolding and pipelining. Clustering is also a viable path to decrease the blowup expansion occuring from the CSDF to HSDF conversion that has not been explored.

Glossary

Actor Atomic part of the application (process), 15

CSDF Cyclo-Static Dataflow, 18

DAG Directed Acyclic Graph, 14

Deadline Extension Amount of slack a task has, 45

Deadline Extension Pool Set containing the amount of slack for every task of the application, 45

Deadlock State from which the application cannot proceed with its execution, 21

Delay Inter-iteration dependency, 16

Execution Time Amount of time a task takes since enabling until completion when execution without interruption, 28

Heracles Static analysis tool, created on NXP Semiconductors and updated as result of this thesis work, 1

Hijdra Hijdra project from NXP Semiconductors, 6

Howard An algorithm to calculate an application's MCM, 32

HSDF Homogeneous Synchrounous Dataflow, 17

MCM Maximum Cycle Mean, 27

MPSoC Multiprocessor System on Chip, 5

NP class Estimation on the complexity of a problem, usually very time consuming to solve, 10

NP-Complete class Estimation on the complexity of a problem, only time consuming, brute force aproaches are known to fully solve such problems, 11

NP-Hard class Estimation on the complexity of a problem, at least as hard as NP problems, sometimes impossible to solve., 11

P class Estimation on the complexity of a problem, usually quick to compute, 10

Repetitions Vector A vector obtained by solving the balance equation of the topology matrix, 19

Response Time Amount of time a task takes since enabling until completion when subjected to processor scheduling interruptions, 28

SDF Synchronous Dataflow, 16

Szymanski An algorithm to calculate an application's MCM, 32

Task Atomic part of the application (process), 15

Token Data on a FIFO buffer, 16

Topology Matrix A compact way to represent a graph, used for conversion amongst different dataflow types, 18

Appendix A

Wlan receptor SDF file descriptor

```
(*router configuration speed 500Mhz, 8 slots -> 500 M slots/s => 500/8 M rots/s *)
(* => latency= 8/500 microsecs = 0.015 microsecs... we rounded up to 1 microsec*10*)
(* Processor types*)
(* 1- EVP*)
(* 2- Sw Decoder*)
(* 3- ARM*)
(* 4- Src*)
(* 5- Snk*)
(* 6- Lat1*)
(* 7- Lat2*)
(* 8- Snk2*)
(* 9- Snk3*)


(* NOTE: Current execution times are all estimations, for a 300MHz EVP.*)
(* Times are given in ns *)
(* Replace all 1 for intended payload size.*)

actors

(*Source Actors*)
name="R320" exec=2400 proct=4 comaptag=1 group=4 slice=2000 map="Src";
name="R321" exec=800 proct=4 comaptag=1 group=4 slice=2000 map="Src";
name="R322" exec=2400 proct=4 comaptag=1 group=4 slice=2000 map="Src";
name="R323" exec=1600 proct=4 comaptag=1 group=4 slice=2000 map="Src";
name="R324" exec=2400 proct=4 comaptag=1 group=4 slice=2000 map="Src";
name="RHeader" exec=4000 proct=4 comaptag=1 group=4 slice=2000 map="Src";
name="RPayload" exec=4000 proct=4 comaptag=1 group=4 slice=2000 map="Src";
name="SIFS" exec=16000 proct=4 comaptag=1 group=4 slice=2000 map="Src";


(*Processing 3500*)
name="Detect" exec=220 proct=1 group=1 slice=450 map="EVP";(*slice=650*)
name="CFEnSync" exec=355 proct=1 group=1 slice=450 map="EVP";
name="FFEnChEst" exec=680 proct=1 group=1  slice=450 map="EVP";
name="HDemode" exec=920 proct=1 group=1 slice=450 map="EVP";
```

```
name="PDemode" exec=920 proct=1 group=1  slice=450 map="EVP";
name="ModHeader"  exec=920 proct=1 group=1 slice=450 map="EVP";
name="AckMod" exec=600 proct=1 group=1 slice=450 map="EVP";

name="PDecode" exec=920 proct=2 group=2 slice=450 map="SwDecoder";
name="HDecode" exec=920 proct=2 group=2 slice=450 map="SwDecoder";(*slice=650*)
name="CodeHeader" exec=920 proct=2 group=2 slice=450 map="SwDecoder";
name="AckCode" exec=600 proct=2 group=2 slice=450 map="SwDecoder";

name="MacCRC" exec=500 proct=3 group=3 slice=900 map="ARM";(*slice=650*)
name="MacAnalyse" exec=1000 proct=3 group=3 slice=900 map="ARM";
name="BuildHeader" exec=500 proct=3 group=3 slice=900 map="ARM";

name="SendPreamble" exec=0 proct=5 group=5 slice=2000 map="Snk";(*slice=2000*)
name="SendHeader" exec=4000 proct=6 group=6 slice=2000 map="Snk2";(*slice=2000*)
name="SendPayload" exec=0 proct=7 group=7 slice=2000 map="Snk3";(*slice=2000*)



(*latency constraint nodes*)
name="LatencyHeader" exec=20000 proct=8 group=8 slice=20000 map="Lat1"; (*slice=20000*)
(* t(startsendpreamble)=2400+800+2400+5*1600+2400+4000+4000*payloadsize+SIFS=36000+4000*payloadsize
(* t(startsendheader)=t(startsendpreamble)+t(sendpreamble) *)
(* t=-t(startsendheader)+mu_d*d-t(sendheader)=-(40000+16000)+40000*2-4000=20000 *)

name="LatencyPayload" exec=20000 proct=9 group=9 slice=20000 map="Lat2";(*slice=20000*)
(*t(startsendpayload)=t(startsendheader)+t(sendheader)*)
(*t=-t(startsendpayload)+mu_d*d -t(sendpayload)=-(40000+20000)+40000*2-0=20000*)

arcs

src="R320" dst="R321" prod=1 cons=1 tokensize=1 delay=0;
src="R321" dst="R322" prod=1 cons=1 tokensize=1 delay=0;


src="R322" dst="R323" prod=5 cons=1 tokensize=1 delay=0;
src="R323" dst="R324" prod=1 cons=5 tokensize=1 delay=0;
src="Detect" dst="CFEnSync" prod=5 cons=1 tokensize=1 delay=0;
src="CFEnSync" dst="FFEnChEst" prod=1 cons=5 tokensize=1 delay=0;

src="CFEnSync" dst="CFEnSync" prod=1 cons=1 tokensize=1 delay=1;

src="R324" dst="RHeader" prod=1 cons=1 tokensize=1 delay=0;

(*Payload size in prod*)
src="RHeader" dst="RPayload" prod=1 cons=1 tokensize=1 delay=0;

src="RPayload" dst="RPayload" prod=1 cons=1 tokensize=1 delay=1;

(*Payload size in cons*)
src="RPayload" dst="SIFS" prod=1 cons=1 tokensize=1 delay=0;

src="SIFS" dst="R320" prod=1 cons=1 tokensize=1 delay=1;
```

```
src="R321" dst="Detect" prod=1 cons=1 tokensize=16 delay=0 <"srrouter" 15 3>;
src="R323" dst="CFEnSync" prod=1 cons=1 tokensize=16 delay=0 <"srrouter" 15 3>;


src="R323" dst="R323" prod=1 cons=1 tokensize=1 delay=1;

src="R324" dst="FFEnChEst" prod=1 cons=1 tokensize=16 delay=0 <"srrouter" 15 3>;

src="FFEnChEst" dst="HDemode" prod=1 cons=1 tokensize=1 delay=0 <"srrouter" 15 3>;

src="RHeader" dst="HDemode" prod=1 cons=1 tokensize=40 delay=0 <"srrouter" 15 3>;
src="HDemode" dst="HDecode" prod=1 cons=1 tokensize=13 delay=0 <"srrouter" 15 3>;

(*Payload Size in prod*)
src="HDecode" dst="PDemode" prod=1 cons=1 tokensize=1 delay=0 <"srrouter" 15 3>;
src="RPayload" dst="PDemode" prod=1 cons=1 tokensize=40 delay=0 <"srrouter" 15 3>;
src="PDemode"  dst="PDecode" prod=1 cons=1 tokensize=45 delay=0 <"srrouter" 15 3>;

(*Payload Size in cons*)
src="PDecode" dst="MacAnalyse" prod=1 cons=1 tokensize=9 delay=0 <"srrouter" 15 3>;

src="HDecode" dst="MacCRC" prod=1 cons=1 delay=0 tokensize=2 <"srrouter" 15 3>;
src="MacCRC" dst="MacAnalyse" prod=1 cons=1 delay=0 tokensize=1 <"srrouter" 15 3>;

(*TODO token size*)
src="MacAnalyse" dst="SendPreamble" prod=1 cons=1 delay=0 tokensize=2 <"srrouter" 15 3>;
src="MacAnalyse"dst="BuildHeader" prod=1 cons=1 delay=0 tokensize=2 <"srrouter" 15 3>;
src="MacAnalyse" dst="AckCode" prod=1 cons=1 delay=0 tokensize=2 <"srrouter" 15 3>;

(*TODO: verify token size*)
src="BuildHeader" dst="CodeHeader" prod=1 cons=1 delay=0 tokensize=2 <"srrouter" 15 3>;
src="CodeHeader" dst="ModHeader" prod=1 cons=1 delay=0 tokensize=4 <"srrouter" 15 3>;
src="ModHeader" dst="SendHeader" prod=1 cons=1 delay=0 tokensize=80 <"srrouter" 15 3>;
src="SendHeader" dst="LatencyHeader" prod=1 cons=1 delay=0 tokensize=0;
src="LatencyHeader" dst="R320" prod=1 cons=1 delay=2;

src="AckCode" dst="AckMod" prod=1 cons=1 delay=0 <"srrouter" 15 3>;
src="AckMod" dst="SendPayload" prod=1 cons=1 delay=0 tokensize=80 <"srrouter" 15 3>;
src="SendPayload" dst="LatencyPayload" prod=1 cons=1 delay=0 tokensize=0;
src="LatencyPayload" dst="R320" prod=1 cons=1 delay=2;
src="SendPreamble" dst="R320" prod=1 cons=1 delay=1;

end
```

Appendix B

TD-SCDMA file descriptor

```
actors

(*mud=675000*)
(* Source has 1.28 MHz freq => T=1/1.28M=781.25 nanos*)
(* Processor types*)
(* 1- EVP*)
(* 2- Sw Decoder*)
(* 3- ARM*)
(* 4- Src*)
(* 5- Latency1*)
(* 6- Latency2*)
(* 7- Latency3*)

name="rx1" exec=275000 proct=4 slice=1000 map="Src" group=4; (* 352 samples*) (*g4*)
name="rx2" exec=12500 proct=4 slice=1000 map="Src" group=4;  (*  16 samples*)
name="rx3" exec=100000 proct=4 slice=1000 map="Src" group=4; (* 128 samples*)
name="rx4" exec=287500 proct=4 slice=1000 map="Src" group=4; (* 368 samples*)


(*tasks*)
name="dass" exec=1820 proct=1 slice=450 map="EVP" group=1;(*exec=1820;*) (*g1*)
name="ce" exec=8610 proct=1 slice=450 map="EVP" group=1;(*exec=8610;*)
name="mi" exec=13280 proct=1 slice=450 map="EVP" group=1;(*exec=13280;*)
name="jd1" exec=129760 proct=1 slice=450 map="EVP" group=1; (*exec=129760;*)
name="jd2" exec=65740 proct=1 slice=450 map="EVP" group=1;(*exec=65740;*)
name="tfci" exec=2000 proct=3 slice=900 map="ARM" group=3; (*exec=2000;*) (*g3*)
name="tpc" exec=2000 proct=3 slice=900 map="ARM" group=3; (*exec=2000;*)
name="decodecrc1" exec=5000 proct=2 slice=450 map="SwDecoder" group=2;(* exec=5000;*) (*g2*)
name="decodecrc2" exec=5000 proct=2 slice=450 map="SwDecoder" group=2; (*exec=5000;*)

name="latency1" exec=285680 (*d=2,mu=675000 L<1062500; t=d.mu -L - tdass*)
proct=5 slice=285680 map="Lat1" group=5;(*g5*)

name="latency2" exec=285500 (*t=d.mu-L-ttpc*) proct=6 slice=285500 map="Lat2" group=6;(*g6*)

name="latency3" exec=395000 proct=7 slice=395000 map="Lat3" group=7;
(*d=10, mu=675000 L<6350000; t=d*mu -L-tdecodecrc2*)(*g7*)
```

```
arcs

src="rx1" dst="rx2";
src="rx2" dst="rx3";
src="rx3" dst="rx4";
src="rx4" dst="rx1" delay=1;
src="rx1" dst="dass";
src="rx2" dst="jd1";
src="rx3" dst="ce";
src="rx4" dst="jd2";

src="ce" dst="dass" prod=1 cons=1 delay=0;

src="ce" dst="mi" prod=1 cons=1 delay=0;
src="ce" dst="jd1" prod=1 cons=1 delay=0;
src="mi" dst="jd1" prod=1 cons=1 delay=0;
src="mi" dst="jd2" prod=1 cons=1 delay=0;
src="jd1" dst="jd2" prod=1 cons=1 delay=0;
src="jd1" dst="tfci" prod=1 cons=1 delay=0;
src="jd2" dst="tpc" prod=1 cons=1 delay=0;
src="jd1" dst="decodecrc1";
src="jd2" dst="decodecrc2";
src="decodecrc1" dst="decodecrc2";
src="decodecrc2" dst="decodecrc1" delay=1;

 (*latencies*)
src="dass" dst="latency1" delay=0;
src="latency1" dst="rx1" delay=2;

src="tpc" dst="latency2" delay=0;
src="latency2" dst="rx1" delay=2;

src="decodecrc2" dst="latency3" delay=0;
src="latency3" dst="rx1" delay=10;


end
```

Appendix C

MPSoC File descriptor

```
processor

name="EVP" wheeltime=1000 type=1 sched="tdma" weight=100 usage=45;
name="SwDecoder" wheeltime=1000 type=2 sched="tdma" weight=100 usage=45;
name="ARM" wheeltime=2000 type=3 sched="tdma" weight=100 usage=45;
name="Src" wheeltime=2000 type=4 sched="off" weight=0;
name="Snk" wheeltime=2000 type=5 sched="off" weight=0;
name="Snk2" wheeltime=2000 type=6 sched="off" weight=0;
name="Snk3" wheeltime=2000 type=7 sched="off" weight=0;

(*latency constraints*)
name="Lat1" wheeltime=20000 type=8 sched="off" weight=0;
name="Lat2" wheeltime=20000 type=9 sched="off" weight=0;

end
```

Bibliography

[1] O. Moreira, Frederico Valente, Marko Bekooij. Scheduling Multiple Independent Hard-Real-Time Jobs on a Heterogeneous Multiprocessor, Proceedings of the 7th ACM & IEEE international conference on Embedded software

[2] Ahmed Amine Jerraya, Wayne Wolf, Multiprocessor Systems-on-Chip. Elsevier, 2005

[3] Jacob Jan David Mol, Resource Allocation for Streaming Applications in Multiprocessors. Delft University of Technology

[4] S. Sriram and S.S. Bhattacharyya. Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker, Inc, 2000.

[5] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. Proceedings of the IEEE, September, 1987

[6] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-Static Dataflow. IEEE TRANSACTIONS ON SIGNAL PROCESSING, VOL. 44, NO. 2, FEBRUARY 1996

[7] Graph Theory An Introductory Course, Bela Bollobas

[8] Bondy, Murty - Graph Theory with Applications [1976, OCR]

[9] Orlando M. Moreira and Marco J. G. Bekooij, Self-Timed Scheduling Analysis for Real-Time Applications, EURASIP Journal on Advances in Signal Processing Volume 2007, Article ID 83710

[10] Marco Bekooij, Orlando Moreira, Peter Poplavko, Bart Mesman, Milan Pastrnak, Jef van Meerbergen Predictable Embedded Multiprocessor System Design. Scopes 2004 workshop, 23 September 2004

[11] A. Dasdan, Experimental analysis of the fastest optimum cycle ratio and mean algorithms, ACM Transactions on Design Automation of Electronic Systems, vol. 9, no. 4, pp. 385418, 2004.

[12] A.H. Ghamarian, et al. Throughput Analysis of Synchronous Data Flow Graphs

[13] R. Govindarajan, Guang R. Gao, Rate-Optimal Schedule for Multi-Rate DSP computations, ACAPS Technical Memo 61, August 16, 1993.

[14] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataow graphs. In Proc. Design Automation Conference (DAC), 2007

[15] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF For Free," in 6th International Conference on Application of Concurrency to System Design, ACSD 2006, Proceedings. IEEE, June 2006, pp. 276-278.

[16] O. Moreira, Jan-David Mol, Marko Bekooij, Jef van MeerBergen, Multiprocessor Resource Allocation for Hard-Real-Time Streaming with a Dynamic Job Mix. In Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium, 2005

[17] Giorgio C. Buttazo, Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, 1997

[18] Sipser, Michael. Introduction to the Theory of Computation, Second Edition. Thompson Course Technology. 2006

[19] Garey Michael R.;Johnson, David S. Computers and Intractability, a guide to the theory of NP-Completeness W. H. Freeman. 1979

[20] Thomas H. Cormen, et al. Introduction to Algorithms, The MIT Press; 2nd edition (September 1, 2001)

[21] J.A. Stankovic. Misconceptions about real-time computing. IEEE Computer, 21(10), October 1988